

AD-A218 168

DTIC FILE COPY

(1)

AN INVESTIGATION OF THE LOCALITY OF MEMORY
ACCESES DURING SYMBOLIC
PROGRAM EXECUTION

by

WILLIAM C. HOBART, JR., B.S., M.S.E.E.

DTIC
ELECTE
FEB 22 1990
S D

DISSSERTATION

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfilment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

THE UNIVERSITY OF TEXAS AT AUSTIN

August, 1989

90 02 21 061

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS NONE			
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION / AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.			
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE					
4. PERFORMING ORGANIZATION REPORT NUMBER(S)		5. MONITORING ORGANIZATION REPORT NUMBER(S) AFIT/CI/CIA- 89-081			
6a. NAME OF PERFORMING ORGANIZATION AFIT STUDENT AT UNIV OF TEXAS AT AUSTIN	6b. OFFICE SYMBOL (<i>If applicable</i>)	7a. NAME OF MONITORING ORGANIZATION AFIT/CIA			
6c. ADDRESS (City, State, and ZIP Code)		7b. ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB OH 45433-6583			
8a. NAME OF FUNDING / SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (<i>If applicable</i>)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER			
8c. ADDRESS (City, State, and ZIP Code)		10. SOURCE OF FUNDING NUMBERS			
		PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (<i>Include Security Classification</i>) (UNCLASSIFIED) An Investigation of the Locality of Memory Accesses During Symbolic Program Execution					
12. PERSONAL AUTHOR(S) William C. Hobart, Jr.					
13a. TYPE OF REPORT X THESIS/DISSERTATION	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 1989 August		15. PAGE COUNT 148	
16. SUPPLEMENTARY NOTATION APPROVED FOR PUBLIC RELEASE IAW AFR 190-1 ERNEST A. HAYGOOD, 1st Lt, USAF Executive Officer, Civilian Institution Programs					
17. COSATI CODES		18. SUBJECT TERMS (<i>Continue on reverse if necessary and identify by block number</i>)			
FIELD	GROUP	SUB-GROUP			
19. ABSTRACT (<i>Continue on reverse if necessary and identify by block number</i>)					
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL ERNEST A. HAYGOOD, 1st Lt, USAF			22b. TELEPHONE (<i>Include Area Code</i>) (513) 255-2259	22c. OFFICE SYMBOL AFIT/CI	

AN INVESTIGATION OF THE LOCALITY OF MEMORY
ACCESES DURING SYMBOLIC
PROGRAM EXECUTION

Accesion For	
NTIS	CRA&I
DTIC	TAB
Unannounced	
Justification	
By _____	
Distribution / _____	
Availability Codes	
Dist	Avail and/or Special
A-	

APPROVED BY

SUPERVISORY COMMITTEE:

W. L. Gray
Dwight Green
Kay
Bassett F. Womack
Marion Gregg
L. A. Clark



To my family

Acknowledgments

I would like to thank my supervising professor, Harvey Cragon, and the other members of my doctoral committee: Les Belady, Mario Gonzalez, David Greene, Roy Jenevein, and Baxter Womack for their guidance and suggestions during my research.

Also, this research would have been much more difficult without the considerable support and assistance of those at Texas Instruments, especially Mark Young, John Osman, Bob Fish, Jim Leftwich, and Mike Fulghum. Their help in modifying the Explorer II¹ system microcode was invaluable in completing this investigation.

I would also like to thank those who provided workloads for this research. At the University of Texas at Austin, I am indebted to Gordon Novak for his GLISP program, Don Simon for his Reducer program, and Ben Kuipers for his QSIM program. I am also grateful to Rick Spickelmier and Karti Mayaram at the University of California at Berkeley for providing the BIASLisp workload.

Finally, I would like to thank those in my research group, especially Yongjae Rim, who provided many helpful comments and suggestions during this research and Joan Van Cleave for her cheerful and efficient administrative support.

WILLIAM C. HOBART, JR.

The University of Texas at Austin

August, 1989

¹Explorer II is a trademark of Texas Instruments, Inc.

**AN INVESTIGATION OF THE LOCALITY OF MEMORY
ACCESES DURING SYMBOLIC
PROGRAM EXECUTION**

Publication No._____

WILLIAM C. HOBART, JR., Ph.D.

The University of Texas at Austin, 1989

Supervising Professor: Harvey G. Cragon

This research focused on the low-level virtual address memory referencing behavior of symbolic programs. The virtual address traces of six artificial intelligence applications and two conventional workloads executed on the Texas Instruments (TI) Explorer II were used to characterize the locality of virtual memory accesses during symbolic program execution and to compare these characteristics with the locality characteristics of conventional workloads. By using this approach, this research accomplished three overall objectives: first, it enhanced the basic understanding of symbolic program memory referencing behavior; second, it found new ways in which the architecture can be tailored for these low-level memory referencing locality characteristics; and third, it laid the foundation for developing a symbolic program workload generator model that can be used to drive memory system design tools.

This research was the first systematic effort to characterize symbolic program behavior through the investigation of symbolic workloads' temporal, spatial,and structural locality. Its specific contributions are the development of new measures and methods for the analysis of program locality, an extension of Denning's model of page referencing behavior to virtual address word-level memory referencing behavior, the identification of a low-level virtual memory referencing structural locality, the identification of significant differences in the low-level virtual memory referencing behavior of symbolic and conventional workloads, and a memory system design for exploiting the structural locality characteristics of symbolic workloads. (AF)

Table of Contents

Acknowledgments	iv
Abstract	v
Table of Contents	vii
List of Tables	xi
List of Figures	xii
1. Introduction	1
1.1 Statement of the Problem	1
1.2 Structure of This Dissertation	2
2. Background	4
2.1 Motivation for This Research	4
2.2 Design Principles for Memory Implementations	5
2.3 Design Alternatives for Memory Implementations	6
2.4 Methods of Quantifying the Locality of Memory Accesses	7
2.5 Past Work Related to This Research	11
2.5.1 Program Modeling	11
2.5.2 Methods Used to Measure Program Locality	14
2.5.3 Published Measurements of Program Locality	18
2.5.4 Proposed Architectures Tailored to Symbolic Workloads	24
2.6 Contributions of This Research	27

2.7 Summary	27
3. Data Collection	29
3.1 Overview of the Experimental Setup	29
3.2 Microcode Modification	31
3.3 Trace Collection Procedure	32
3.4 Workload Selection	33
3.4.1 Selection Criteria	33
3.4.2 Modifications Made to the Workloads	35
3.5 Extraction of the Sample Data from the Workload Traces	35
3.5.1 Determination of the Sample Length	35
3.5.2 Extraction of the 450,000 Reference Sample	36
3.6 Summary	36
4. Trace Analysis	40
4.1 Markov Model of Low-Level Memory Referencing Behavior	40
4.2 Trace Categorization and Data Compression	43
4.3 Locality Characteristics Computed	44
4.4 Summary	50
5. Results	51
5.1 Spatial Locality	51
5.1.1 Existence of a Spatial Locality Window	51
5.1.2 Differences in P_{SW} for Symbolic and Conventional Workloads .	53
5.1.3 Implications for Symbolic Memory Subsystem Design	55
5.2 Temporal Locality	55
5.2.1 LRU Stack Distance Thresholds	55

5.2.2	Stack Distance Thresholds for Symbolic and Conventional Workloads	56
5.2.3	Implications for Symbolic Memory System Design	69
5.3	Structural Locality	69
5.3.1	Structural Locality Metric	69
5.3.2	Differences in Symbolic and Conventional Workload Structural Locality	70
5.3.3	Implications for Symbolic Memory System Design	71
5.4	State Transition Probabilities	72
5.4.1	Metrics Used for the Comparison	72
5.4.2	Differences Between Symbolic and Conventional Workloads	72
5.4.3	Implications for Symbolic Memory Subsystem Design	75
5.5	Results of Correlogram and Power Spectrum Analysis	75
5.6	Summary	80
6.	Validation of Results	83
6.1	Validation of the Software Routines	83
6.2	Evaluation of Architectural Independence of Results	84
6.2.1	Distinctive Architectural Features of the Explorer II	85
6.2.2	Comparison Between Explorer II and IBM System/360 Model 91 Traces	88
6.3	Summary	94
7.	Application to Memory Subsystem Design	96
7.1	Motivation for the Design	96
7.2	Proposed Design	97
7.3	Specific Design Parameters	99

7.4	Analytic Model of Effective Memory Access Time	99
7.4.1	CAM Cache Hit Probability and Design Parameters	100
7.4.2	Structural Locality Cache Hit Probability and Design Parameters	107
7.5	Performance Analysis	112
7.6	Summary	121
8.	Conclusion	122
8.1	Main Contributions	122
8.2	Additional Applications of This Research	123
A.	Individual Workload Locality Measurements	126
BIBLIOGRAPHY		142
Vita		

List of Tables

3.1 Explorer II Workloads Traced	34
4.1 Relative Frequency of Types of References	43
5.1 Spatial Window Probability (P_{SW})	53
5.2 LRU_{90} Stack Distance Thresholds	67
5.3 LRU_{95} Stack Distance Thresholds	67
5.4 LRU_{99} Stack Distance Thresholds	68
5.5 P_{SSD} Transition Probabilities	70
5.6 P_{ON} Transition Probabilities	73
5.7 P_{NO} Transition Probabilities	74
7.1 Memory Subsystem Design Parameters	99
7.2 Structural Locality Cache Hit Probability (p_{SLC})	112
7.3 Performance Analysis for $R_{CAM} = 4$ and $R_{MEM} = 32$	113

List of Figures

2.1 Spatial Locality	8
2.2 Temporal and Structural Locality	8
3.1 Project Overview	30
3.2 Example Spatial Locality Histogram for a 10,000 Reference TMYCIN Sample	37
3.3 Example Spatial Locality Histogram for a 100,000 Reference TMYCIN Sample	38
3.4 Example Spatial Locality Histogram for a 450,000 Reference TMYCIN Sample	39
4.1 Two-State Markov Model of Program Behavior	41
4.2 Three-State Markov Model of Program Behavior	42
4.3 Example Individual Frequency Histogram	46
4.4 Example Cumulative Frequency Histogram	47
4.5 Example Correlogram	48
4.6 Example Power Spectrum	49
5.1 BIASLisp Cumulative Temporal Locality Histogram	57
5.2 FFT Cumulative Temporal Locality Histogram	58
5.3 Boyer Cumulative Temporal Locality Histogram	59

5.4	Compile-RB Cumulative Temporal Locality Histogram	60
5.5	Compile-Str Cumulative Temporal Locality Histogram	61
5.6	GLISP-Comp Cumulative Temporal Locality Histogram	62
5.7	GLISP-Pay Cumulative Temporal Locality Histogram	63
5.8	QSIM Cumulative Temporal Locality Histogram	64
5.9	Reducer Cumulative Temporal Locality Histogram	65
5.10	TMYCIN Cumulative Temporal Locality Histogram	66
5.11	Example Correlogram of a Spatial Distance String	76
5.12	Example Power Spectrum of a Spatial Distance String	77
5.13	Example Correlogram of a Temporal Distance String	78
5.14	Example Power Spectrum of a Temporal Distance String	79
5.15	Correlogram of the FFT Data Write New-New Spatial Distance String	81
5.16	Power Spectrum of the FFT Data Write New-New Spatial Distance String	82
6.1	IBM FFT1 Cumulative Spatial Locality Histogram	89
6.2	IBM FFT2 Cumulative Spatial Locality Histogram	90
6.3	IBM APL-Plotter Cumulative Spatial Locality Histogram	91
6.4	IBM WATFIV Compiler Cumulative Spatial Locality Histogram	92
6.5	IBM WATEX Bin Packing Cumulative Spatial Locality Histogram	93
7.1	Proposed Memory Subsystem Design	98

7.2	Markov Model for Main CAM Cache Referencing with No Prefetching	101
7.3	W_c vs. Effective Cache Size	103
7.4	Markov Model for Main CAM Cache Referencing with Prefetching . .	106
7.5	Markov Model for On-Chip SLC Referencing with No Prefetching . . .	108
7.6	W_s vs. Effective On-Chip SLC Cache Size	109
7.7	Markov Model for On-Chip SLC Referencing with Prefetching	110
7.8	Speedup vs. R_{CAM} and R_{MEM} for $N_S = 2$ words	114
7.9	Speedup vs. R_{CAM} and R_{MEM} for $N_S = 4$ words	115
7.10	Speedup vs. R_{CAM} and R_{MEM} for $N_S = 8$ words	116
7.11	Speedup vs. R_{CAM} and R_{MEM} for $N_S = 16$ words	117
7.12	Speedup vs. R_{CAM} and R_{MEM} for $N_S = 32$ words	118
7.13	Speedup vs. R_{CAM} and R_{MEM} for $N_S = 64$ words	119
7.14	Speedup vs. R_{CAM} and R_{MEM} for $N_S = 128$ words	120

Chapter 1

Introduction

1.1 Statement of the Problem

Ideally, memory subsystem design should be a systematic process which results in a memory that conforms to the constraints imposed by other computer subsystems and which exploits the characteristics of its expected workloads. However, current memory architecture designs for symbolic processing systems either are optimized for the memory reference localities observed in conventional workloads or for the memory access characteristics that are assumed for symbolic workloads based on a high-level analysis of symbolic programs. Much more research into the memory access characteristics of symbolic workloads is required before cache memory design and paging algorithms can be optimized for symbolic processing workloads.

This, then, was the purpose of my research:

- To further the characterization of memory accesses during symbolic program execution, and in so doing,
- To progress toward a more systematic memory subsystem design procedure.

In accomplishing this goal, this research also:

- Developed new measures, models, and methods for the analysis of program locality,

- Identified the specific low-level memory referencing behavior associated with the previously little understood structural aspect of program locality, and
- Provided a systematic analysis of a novel design alternative for exploiting the special locality characteristics of symbolic workloads.

1.2 Structure of This Dissertation

The next chapter provides the context for this research. It begins by establishing the need in memory subsystem design for an adequate characterization of workload memory referencing behavior. The chapter then summarizes other work done in the characterization of memory referencing behavior as well as other attempts to exploit program locality in the memory subsystem design. Chapter 2 ends with a description of the scope of the research in this dissertation and how it relates to the other work summarized in this chapter.

Chapter 3 describes the collection of the virtual memory address traces. This description includes the experimental setup, the selection of the workloads, and the selection of the trace segments used in the trace analysis.

Chapter 4 then describes the procedure used to analyze these traces. The chapter starts with an explanation and justification of a Markov model of program behavior upon which much of the analysis is based. Then, using this Markov model, the data compression and sorting techniques used are described, as well as each of the locality characteristics computed for the workload traces.

The results of the trace analysis are described in Chapter 5. This chapter characterizes the spatial, temporal, and structural locality of symbolic workloads and contrasts these locality characteristics with those of conventional workloads. In this chapter, these differences are not only statistically quantified, but are also explained in the light of previously published research and the known high-level behavior of

symbolic workloads. Finally, Chapter 5 concludes with an analysis using the Markov model developed in the previous chapter to show further differences between symbolic and conventional workloads in their low-level memory referencing behavior.

Chapter 6 describes the procedures used to validate the results of the previous chapter and assesses the architectural independence of the results presented in Chapter 5. The validation procedures include numerous cross-checks for consistency among the results and comparison with virtual memory address traces generated on an IBM System/360 Model 91. To assess the architectural independence of the results, the influence of each of the distinctive architectural features of the Explorer II on the virtual address memory referencing behavior are evaluated in this chapter.

Chapter 7 provides an example of applying the results of this research by evaluating a novel memory subsystem design also resulting from this research. This evaluation develops and uses an analytic model of the processor-to-memory effective access time for this design to select the cache design parameter values which will minimize the effective memory access time for the measured symbolic workload locality characteristics. This candidate design is then compared to the existing Explorer II memory design and shown to provide over a 20 percent speedup in the effective memory access time for symbolic workloads.

Finally, Chapter 8 summarizes the major contributions of this research and suggests additional applications of the results of this research.

Chapter 2

Background

This chapter begins by explaining how workload characterization affects memory system design. Next, the related research that has been accomplished in the characterization and exploitation of program memory referencing locality is summarized, thus giving the context for my research. Because so much research has been done in this area and because so many approaches have been used in measuring program locality, any summary of the published research in this area is by necessity extensive. This chapter is no exception. To enhance the readability of this summary, related results are grouped together and cross-referenced, rather than simply listing the contributions in chronological order.

2.1 Motivation for This Research

One of the key input parameters in designing a computer memory subsystem is the characterization of the expected workload. And, one of the key workload features is the word-level virtual memory referencing behavior during workload execution. For this reason, researchers have extensively studied these memory access patterns for both scientific and data processing workloads. The characterizations of these workloads have then been used to exploit observed spatial and temporal locality with cache memory implementations, paging algorithms, and instruction branch prediction strategies.

However, in comparison to the characterizations of scientific and data pro-

cessing workloads, the memory access characteristics of symbolic processing workloads are relatively unknown. But characterizing symbolic workloads is essential if three important questions are to be answered. First, are there any basic differences in the locality characteristics of conventional and symbolic workloads that would justify a memory subsystem design tailored to symbolic workloads? Second, what are the differences and how should the memory design be tailored? And, finally, how does the performance of a memory subsystem tailored for symbolic workload execution compare with the performance of a conventional memory subsystem design?

In addition to answering these questions, this research lays the foundation for further investigation of these differences and provides a basis for evaluating new memory designs that claim to be tailored for symbolic processing computer systems. And, finally, this characterization provides a basis for developing symbolic workload generator models and so aids in the development of expert systems that guide memory subsystem design.

2.2 Design Principles for Memory Implementations

In designing a memory subsystem implementation, the principal objective is to minimize the time required for processor reads and writes while staying within the constraints imposed by cost and physical size. While the actual effective memory access times depend greatly on the technology used to realize a memory implementation, the designer seeks an implementation that will make best use of a technology once it is selected. The effective memory access time will also vary with the type of workload being executed by the computer system. If the computer system is being designed for a particular application, the designer should optimize the memory implementation for this type of workload. This is done by first characterizing the memory referencing locality of this type of workload, and then exploiting these

locality characteristics to decrease the effective access time of the memory system.

A secondary factor which may also have to be considered is the processor-to-memory bandwidth required to allow the processor to run at its maximum speed. Many design alternatives which decrease the effective memory access time, do so at the expense of increasing the required processor-to-memory bandwidth; and so, this tradeoff must be addressed.

2.3 Design Alternatives for Memory Implementations

There are many design alternatives for memory implementations that can decrease the effective memory access time. For example, if the workload tends to concentrate its accesses in a small subset of memory locations, a small high-speed cache memory may greatly decrease the effective memory access time. If the cost of the memory subsystem is kept constant, then, in most cases, a hierarchical memory system consisting of two or more levels will improve the overall access time of the memory subsystem when compared to a single-level memory subsystem. The number of levels, as well as their speed and size ratios; whether demand or prefetching strategies are used; the replacement and write back algorithms chosen; and sizes of the blocks of data transferred between these levels of memory are all design alternatives that will affect the memory system performance. Other design alternatives include the choice of what memory contents to cache and how to map the addresses of the lowest level memory into the higher levels of memory.

If on-chip memory is included in the memory hierarchy, then still more design options are available. The on-chip memory can be used in place of or in combination with general purpose registers, as an instruction cache, as a data cache, or as a top of stack (TOS) buffer used to cache both instructions and data.

2.4 Methods of Quantifying the Locality of Memory Accesses

The primary ways of quantifying the locality of memory accesses center on the concepts of spatial and temporal locality. A workload exhibits spatial locality if reference to a memory location increases the probability that subsequent accesses will be to nearby memory locations. In contrast, a workload exhibits temporal locality if reference to a memory location increases the probability that it will be referenced in the near future.

In addition to these concepts there is a third type of locality. A workload exhibits this third type of locality, which Thazhuthaveetil has termed structural locality, if reference to a particular memory location increases the probability of a subsequent access to another specified memory location [Thaz86]. In other words, these memory locations seemed to be paired together in some fashion such as belonging to the same structure. These three types of locality are summarized below:

Spatial Locality infers that the subsequent reference is likely to be to a location near the current reference in the virtual memory address space

Temporal Locality infers that the subsequent reference is likely to be to one of those addresses referenced in the recent past

Structural Locality infers the subsequent reference will be to address B given that the program has just referenced address A and such that the previous reference to address A was followed by a reference to address B

Figures 2.1. and 2.2. illustrate these three types of locality.

The spatial, temporal, and structural aspects of locality are, to a large extent, independent of each other. For example, a program consisting of a few large subroutines each with no branch instructions would be expected to have high spatial

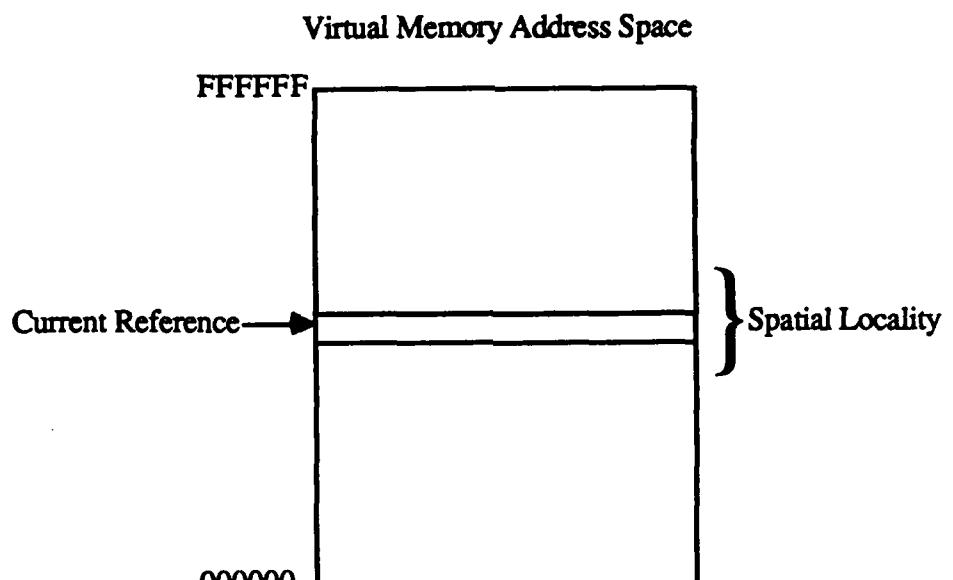


Figure 2.1: Spatial Locality

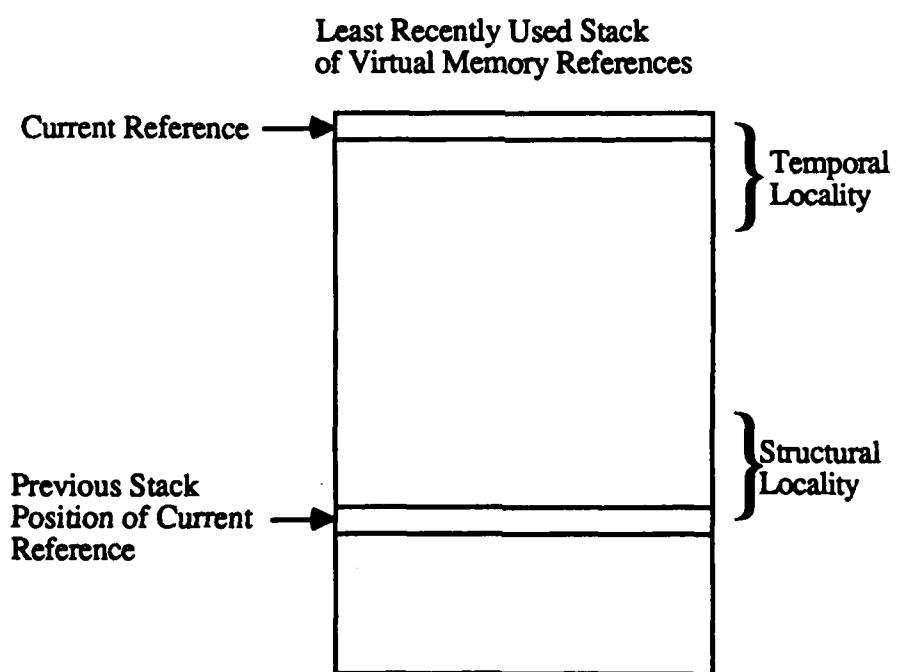


Figure 2.2: Temporal and Structural Locality

locality for the instruction stream since consecutive instructions are normally stored in consecutive virtual memory addresses. However, this program's instruction stream would have little temporal locality since the large size of the subroutines and the lack of looping within the subroutines would make the re-referencing of an instruction in the near future unlikely. Finally, the instruction stream would have structural locality since the re-referencing of a subroutine would provide a high probability that other instructions in that subroutine would be referenced in the near future. Likewise, other examples can be constructed to show that virtual memory address streams can exhibit any combination of spatial, temporal, and structural locality.

Most of the work in measuring the locality of memory references focuses on the temporal aspect. Denning proposed the most common method of quantifying temporal locality in a multiprogramming environment: defining a working set of the program as the set of memory pages that were referenced by the program during the previous T memory references where T is called the window size [Denn68]. The working set attempts to characterize the temporal locality that exists at each point in time and this is then used as a predictor for future memory references. A number of attempts have been made to model the memory reference pattern of conventional workloads mathematically. A semi-Markov model for the univariate point process of page exceptions was proposed by Lewis and Shedler and provides a fairly accurate characterization of the paging behavior for the workloads they executed [Lewi73].

Spatial locality for instruction fetches is sometimes expressed as the frequency of successful branches, the distribution of branch distances, or the average instruction loop length [Alex75]. For symbolic workloads, Clark has measured both the static and dynamic spatial locality of list structures by tabulating histograms of the pointer distances of list cells [Clar77, Clar79].

Thazhuthaveetil defined list sets in a way analogous to working sets and

then used the size of the list set and its lifetime as measures of a program's structural locality. He defined a list set as the union of a list reference with the list sets of its *car* and *cdr* provided that the references to the *car* and *cdr* were temporally adjacent. Thazhuthaveetil used ten percent of the length of the address trace as the maximum separation of two list references that could be considered to be temporally adjacent. He defined the lifetime of the list set as the number of references between the first reference to the list set and the last reference to it. Then, using these characteristics, Thazhuthaveetil partitioned the reference string into a set of possibly overlapping localities just as the working set does for conventional workloads. He also proposed a metric for static structural locality that characterizes the effective branching factor of lists [Thaz86].

However, there are no universally accepted metrics for comparing the dynamic spatial, temporal, and structural localities of programs. Instead, the most common approach is to infer the locality of the workload from the cache miss ratio or the page fault rate for a given implementation of a memory system. A. Smith has been a main contributor in this area by using address traces to drive simulations and then measuring the effects on the memory system performance as the design parameters are varied [Smit85a]. However, using this procedure does not produce an absolute quantification metric; rather, only the relative localities of the two workloads can be inferred. Furthermore, this approach is not a preferred design methodology, since with the many parameters in a memory subsystem design, it is extremely unlikely that the optimal n-tuple of these parameters in the design space can be found using this approach.

2.5 Past Work Related to This Research

2.5.1 Program Modeling

The initial research in this area in the 1960's concentrated on developing more efficient paging algorithms for multi-programmed computer systems executing conventional data processing workloads. Belady performed much of the initial work and developed the MIN algorithm for assessing paging performance [Bela66]. This algorithm assumes prior knowledge of future memory references and is thus not feasible to implement. However, because it is an optimal algorithm (minimizing the page fault rate for a given memory reference string), it serves as a standard for determining how closely other implementable paging algorithms approach optimality.

Belady and Kuehner then proposed a lifetime function which relates the amount of memory allocated to a program to the average number of memory references it makes between page faults [Bela69a]. This function is *S*-shaped with the convex portion following the curve $e = as^k$ where e is the average number of memory references between page faults, a is a constant, s is the storage allocated to the program, and k is approximately equal to two.

Olsson, in 1981, measured the paging performance of an InterLisp system implemented in FORTRAN and hypothesized that the lifetime function proposed by Belady and Kuehner could better be represented as a family of lifetime functions when modeling the paging performance of symbolic workloads [Olss83]. This family of lifetime functions has the characteristic shape of the Belady lifetime function, but the slope and tail-off of each function is dependent on the page size as well as the amount of memory allocated. However, Olsson acknowledges the limited amount of data upon which his hypothesis is based and recommends further measurements of this type be taken to confirm or reject his hypothesis.

Mattson, Gecsei, Slutz, and Traiger in 1970 used address traces to evaluate replacement algorithms and the effect of memory allocation on the page fault rate [Matt70]. They specifically addressed a class of algorithms which met their definition of a stack algorithm and then used a least recently used (LRU) stack model for their evaluation of the address traces. They defined a metric, stack distance, as the number of pages referenced since the current page was last referenced and assigned a stack distance of infinity to any page that was referenced for the first time. Using these stack distances, they computed a distance string from the address trace which could then be used in one pass to determine the page fault rate iteratively for all possible memory allocations.

Denning states in [Denn72] that

the property of locality can be summarized as three statements:

1. A program distributes its references non-uniformly over its pages, some pages being favored over others.
2. The density of references to a given page tends to change slowly in time.
3. Two reference string segments are highly correlated when the interval between them is small, and tend to become uncorrelated as the interval between them becomes large.

He also proposed modeling the program behavior as transitions through a collection of overlapping localities. This view of program behavior is still a basis for workload characterization today.

As stated in a previous section, Lewis and Shedler in [Lewi73] have developed a micromodel of sequences of page faults from statistical modeling of address traces. They call their model a micromodel because its domain is restricted

to steady state program execution and does not apply to the loading of the initial working set of a program. Since the occurrence of a page fault is relatively rare once the initial working set is loaded, Lewis and Shedler first attempted to model the sequences of page exceptions as a Poisson process. However this model proved to be highly inaccurate and a two-state semi-Markov model was next tried. This model had to duplicate the high degree of skewness in the empirical data, and thus, a mixed geometric and negative binomial plus one probability distribution was chosen. The estimation procedure for the parameters of the distribution was ad hoc by the authors' own admission and in the final analysis, the authors suggested that a three-state semi-Markov process might be a more accurate model. The third state would model a referencing mode that they encountered where a very large number of memory locations would be accessed consecutively with page faults occurring only at the page boundaries.

Spirn has suggested that one of the states in Lewis and Shedler's model may be due to the referencing within a locality and the other state due to referencing that takes place when the locality is changed [Spir77]. In this second mode, page exceptions are much less rare, and it appears that it is the presence of this relatively rapid transition between localities that causes the Poisson process model to be inadequate.

J.E. Smith and Goodman studied cache organizations and replacement policies for instruction caches [Smit83]. In this research, they used analytical models of instruction referencing patterns to indicate design choices for an instruction cache design. To model instruction fetching, Goodman used a loop model and a complex loop model. In the complex model, consecutive instruction fetches are not necessarily made to consecutive memory addresses. Using these analytical models, they show for the simple loop model that the random replacement algorithm is superior to the

LRU and First In First Out (FIFO) algorithms. They also show that a direct-mapped cache is superior to a fully associative cache for LRU and FIFO algorithms but that the two cache designs have equivalent performance for the random replacement algorithm. Finally, they conclude that a set-associative cache with random replacement combines the advantages of the direct-mapped and fully-associative caches for the simple loop model. No significant conclusion could be derived, however, for the complex loop model. Finally, they determined that both cache organization and replacement policies are secondary factors and that the primary factor controlling cache performance is its size.

2.5.2 Methods Used to Measure Program Locality

In the mid-1970's, Alexander and Wortman published their analysis of the static and dynamic characteristics of instruction fetches in XPL programs [Alex75]. To gather their data, they modified the microcode of an IBM System/360 to implement a technique known as jump tracing. This trace technique records only the memory addresses referenced by the branch instructions and thus greatly reduces the amount of trace data to analyzed. To determine the static characteristics, they modified the XPL compiler that they were using. Among the static characteristics they gathered were the frequency of occurrence of each instruction in the program and the relative frequency of operators. They were also able to measure the relative frequency with which instructions were executed and to determine the distribution of branch distances. These statistics continue to be referenced as a basis for conventional instruction set design and understanding program behavior.

As cache memories became more common in the late 1960's and early 1970's, research into better cache designs and replacement algorithms increased. This research built upon the knowledge used to develop paging systems. As mentioned in

the previous section, A. Smith introduced the use of trace-driven simulation in the late 1970's to evaluate cache design alternatives and many other researchers have built upon his work. Also, in 1977, Smith proposed two methods of reducing the trace data for analysis [Smit77]. The first method, which he called Stack Deletion, reduced the trace by deleting all memory references which were rereferences to one of the last $k - 1$ memory locations referenced where k is defined as the stack level. For instance, if $k = 1$, then all immediate rereferences would be deleted from the trace. Smith's other proposed method was the Snapshot method. In the Snapshot method, a routine periodically interrupts the program being traced and records those pages whose page reference bits are currently set. By scanning the page reference bits in a pseudo-random order, the LRU stack can be replicated and the page fault rate predicted with a high degree of accuracy. Both of these methods have the potential of reducing the trace data by one to two orders of magnitude and thus allow the analysis of longer trace intervals.

Clark and Emer built a micro-PC histogram monitor which allowed the relative frequency of various architectural events, such as the execution of specific opcodes, to be determined [Emer84]. Because they used a separate passive hardware monitor on the VAX Unibus, they were able to trace the execution of the workloads at full speed and thus compile data for up to two hours of CPU processing at a time. However, because they were strictly counting the total number of each event that occurred during that time, they had no way of observing changes that occurred in the relative frequency of events during that two-hour interval. There were also several architectural events that they could not monitor because the hardware signals were not available on the Unibus.

In [Smit85a], A. Smith points out six drawbacks of trace-driven simulation. First, the trace must necessarily represent only a very small fraction of the

actual workload. Second, it is very difficult if not impossible to guarantee that this snapshot is typical of the overall workload, and, in practice, the part of the workload traced has better locality than average. Third, the traces do not usually include the memory references due to the operating system, even though studies such as Clark and Emer's [Clar85] have shown the operating system to have a great effect on the system performance. Fourth, task switching and interrupt servicing are usually not included in the address traces and so their effects on system performance are masked. Fifth, the trace data is influenced by the implementation of the architecture and so the order of the memory references will differ from one implementation to another. Finally, most traces do not capture the effect of input/output on the cache activity.

Agarwal, Sites, and Horowitz proposed a new technique for capturing address traces by modifying the microcode on the VAX 8200 to record memory references as they are made [Agar86]. They state that this technique has the advantage of slowing down the execution by only one order of magnitude rather two or three orders of magnitude as is the case when an operating system trace mode is used or when a software simulator generates the trace. Also, the virtual memory addresses are recorded rather than physical addresses thus eliminating the distortion of implementation features such as instruction prefetch buffers. Further, since every memory reference is recorded, the trace does not suffer from the granularity distortion that can limit the usefulness of traces obtained from modifications to the page fault handler that record only page exceptions. As they point out, there is no omission distortion, since all memory references due to interrupt servicing, task switching, and operating system calls can also be recorded. Other advantages include being able to record other CPU internal state information along with the virtual address and not requiring additional hardware to record the trace data. Because they modified each microcode location that could request a memory reference, they had about 80 distinct microcode modifications and were not able to modify all necessary microcode

locations due to exhausting the spare microcode memory. They found through their tracing that with the inclusion of system references in the address traces, that for caches under 256K bytes, the cache size had to quadruple to have the same miss ratio achieved with the address trace of the user program by itself. They were also able to evaluate the effect of process switching and found that for caches smaller than 16 Kbytes, multiprogramming increased the cache miss rate only slightly over the miss rate when the programs were run sequentially in a uniprogramming environment and that the small increase could be modeled with cache purges. For cache sizes above 64K bytes, little was gained from cache purges and a cache greater than 256K bytes was able to hold the working set of several processes simultaneously.

Trace flattening, developed by McNiven and Davidson, is the most recently used method of measuring program locality [McNi88]. This technique traces values rather than memory locations. Thus, a memory location which is used at different times in the program for different values is treated as a separate entity for each value. Likewise, different memory locations holding the same value as a result of a move instruction are treated as one entity as long as they hold the same value. McNiven and Davidson state that trace flattening masks the effects of the architecture and compiler by integrating the register set into the memory hierarchy and by simulating an ideal compiler that does not needlessly move data. They also introduce the concept of excess traffic, which is the traffic generated by having to reread data which was previously in the local memory. Using trace flattening and the concept of excess traffic, McNiven and Davidson divided the values into classes based on their interreference times and lifetimes. These classes of values were then individually analyzed and appropriate memory caching strategies developed for each class.

2.5.3 Published Measurements of Program Locality

In 1977, Clark and Green published their analysis of the static spatial locality of the list structure data produced by Lisp programs. They found substantial spatial locality and little sharing of list structures [Clar77]. Another interesting result was that they found that the frequency with which atoms were referenced basically followed Zipf's Law [Zipf49]. That is, the n th most common atom was referenced $1/n$ as many times as the most common atom. They also found that static spatial locality between successive list elements was primarily due to the elements being created close together in time and that using sophisticated storage allocation algorithms for list element creation improved the static spatial locality by only a few percent. They also quantified the improvements in static spatial locality of the list data structures after recopying the list structures to make successive list elements follow each other in memory whenever possible, a process called list linearization. Finally, they proposed encoding the pointer from a list element to its successor with an offset code, laying the foundation for *cdr*-coding.

In 1979, Clark published measurements of the dynamic spatial locality of symbolic programs including CONGEN, an expert system to generate chemical structures; NOAH, a hierarchical planning system; and SPARSER, a speech understanding parser [Clar79]. In his measurements, the frequency with which atoms were dynamically referenced dropped off much faster than Zipf's Law indicating a more uneven distribution of atom references. However, only a small portion of each program's execution could be traced without having unreasonably long traces. Clark conjectures that this limitation caused the program to be traced while it was only in a few of its localities and offers this explanation for the differences between the static and dynamic frequency of reference measurements. Clark found that the primitive functions *car* and *cdr* accounted for the great majority of the function executions.

He also found that the *nil* primitive function was executed less dynamically than it was present statically which would seem to infer that lists are usually not traversed to their end. He confirmed that the static spatial locality that he had previously measured was indicative of a dynamic spatial locality. In fact, the percentage of list pointers pointing to the same page was basically equivalent for the static and dynamic measurements. Finally, he determined that once a list was linearized, it tended to remain linearized.

Foderaro and Fateman added to the characterization of symbolic workloads through their measurements of Macsyma on the VAX-11 in the FranzLisp environment [Fode81]. They validated a design decision to treat the garbage collection phase as an abnormal memory referencing mode by rebuilding the working set from scratch once the garbage collection phase had ended. They also found for their demonstration scripts that 385 cons cells were allocated and quickly deallocated for each 'permanent' cons cell that was allocated. However, when bignums (arbitrarily precise integers represented by linked lists) were calculated in a program computing the Binet function, one 'permanent' cons cell was created for every 37 temporary cons cells created. In another demonstration script, the Begin demo, they found that the functions *cons* and *equal* accounted for a significant portion of the instructions executed and that the most common VAX-11 assembly language instruction was *movl* (move longword) with a static frequency of occurrence of 43 percent and a dynamic frequency of occurrence of 27 percent. They also found that integer arithmetic was used very little. These latter results are roughly comparable to the relative instruction frequencies of XPL programs compiled by Alexander and Wortman.

Goodman studied the design of on-chip data caches which are often constrained by the bandwidth available for off-chip memory references [Good83]. He showed that while larger block sizes exploit spatial locality in cache design, that

they can greatly increase processor-memory traffic. In systems where the processor-memory bandwidth is the primary limiting factor on system performance, smaller block sizes should be used and thus temporal locality should be primarily exploited. He also determined that the trace results were easier to interpret if the parameters were plotted after averaging them over all the traces to eliminate individual program anomalies. He found that spatial locality dominated the cold start period when the working set for a program was being loaded, while temporal locality dominated the warm start period (the period after the initial working set was loaded). He also proposed the use of transfer blocks that were sub-blocks of the address block. This reduces the amount of memory required for the tags through the larger address block size while decreasing the processor-memory traffic through the smaller transfer block size.

Stanley and Wedig have also studied the locality of on-chip caches. In [Stan87], they measured the performance of various TOS buffer management algorithms for an on-chip data cache. Using the Dhrystone benchmark on the VAX-11, they noted that TOS buffers only 16 elements in size cut the processor-to-memory data stream bandwidth in half. They attributed this to the very high temporal and spatial locality of the data references.

However, Goodman and Hsu contend that available on-chip memory can be more effective in reducing the required memory bandwidth if the memory is organized as registers and a sophisticated register allocation scheme is used, rather than if the available on-chip memory is organized as a cache [Good86]. In their trace-driven simulation of five UNIX traces on the VAX-11, they found that the bus traffic for the register organization was consistently less than that for the LRU cache for on-chip memory sizes of 2 to 64 words. For example, with a memory size of 16 words, the bus traffic was decreased by 68 percent when organized as registers instead of an

LRU cache.

On the other hand, Flynn and Mitchell have found that, in most cases, a single register set used with a data cache is more effective in reducing the data traffic than approaches using all available chip area for multiple register sets [Flyn87]. Additionally Mitchell, in his modeling of processor architectures, showed that a stack architecture requires a 3 Kbyte instruction cache to have the same instruction stream memory traffic as an IBM System/360 register architecture with a 1 Kbyte instruction cache [Mitc86]. Thus, they found it necessary to adopt an approach, combining both instruction set and memory subsystem design, that makes balanced reductions in both instruction and data traffic.

Goodman and Chiang found in their trace-driven simulations of conventional workloads on the VAX-11 that instruction dynamic spatial locality was much greater than the data dynamic spatial locality [Good84]. They also noticed that as amount of memory in computers has increased, the locality of the programs written on the computer has correspondingly decreased as programmers exploit the extra memory available. Finally, they concluded that the static column RAM cache that they were analyzing would be more effective caching both instructions and data than if it was used to cache only instructions with no cache for the data references.

Using their trace flattening technique mentioned in the previous section, McNiven and Davidson found that most of the bus traffic is caused by great number of values having very short lives and the values with longer lives that are referenced frequently. They state that the great number of short-lived values is why a fairly small number of registers so effectively reduces the memory bus traffic, and they recommend using compiler-generated information that identifies values which will no longer be referenced (dead values) to enhance the cache replacement algorithm.

Clark and Emer determined from their VAX measurements in [Emer84]

that a small number of complex instructions accounted for 90 percent of the CPU execution time and a large number of the memory references even though 84 percent of all instructions were moves, branches, and other simple instructions. They also determined that the average number of loop iterations was 10 and that branches and subroutine calls accounted for about 40 percent of the instructions that were executed. They measured the relative frequency of addressing modes, and also found the average ratio of reads to writes to be about two to one.

In [Smit85a] A. Smith attempted to minimize the error due to the first three drawbacks of trace-driven simulation mentioned in Section 2.5.2 by using a large number of workloads (49) and using traces gathered with a hardware monitor that included memory references due to operating system calls. Significantly, for my research, he included in his collection of workloads several Lisp program traces of a Lisp compiler and Vaxima runs to represent symbolic workloads. Contrary to his expectations, his measurements revealed little difference in the program statistics for conventional and symbolic workloads on the VAX-11/780. He found that the ratio of reads to writes was about two to one corroborating the findings of Clark and Emer. He also found that the percentage of data reads was very stable across the various workloads while the percentage of data writes varied widely. He was able to postulate a few rules of thumb for the IBM System/370 and VAX workloads. One such rule was that half the memory references are instruction fetches. Another rule of thumb was that approximately half of the data lines being pushed out of the cache are 'dirty', thus requiring a write-back to memory. A. Smith found that the data space for most workloads was larger than the instruction space and that there was less locality in data than in instructions. Finally, Smith pointed out the need for additional traces for both the machines traced in his research as well as other machines, and he identified the need for improving the quality of the traces by

obtaining traces that are consistent with the assumptions made in his paper about instruction buffering and the quality of the compilers used.

At the lowest end of the memory hierarchy, A. Smith has used the trace-driven simulation technique to analyze the design and effectiveness of various disk cache implementations [Smit85b]. In this work, he found that the main memory was better used by the paging algorithms than by setting aside a portion of it as a specialized disk cache.

Hill and A. Smith further studied the design of on-chip microprocessor caches through trace-driven simulation using conventional workloads [Hill84]. They analyzed the contributions of a minimum cache and of a smart cache that exploits the instruction and data referencing patterns. Through their simulation results, they concluded that a small minimum cache can be very effective, especially when used as an instruction buffer. For the smart cache, they implemented a load-forward mechanism where only the sub-blocks forward of the referenced location in the address block were prefetched. While the load-forward mechanism increased the miss ratio by 7 percent, the processor-memory traffic ratio was reduced by 20 percent. Thus, this prefetch algorithm could be useful where the bus traffic is a limiting factor on system performance.

Lee and A. Smith have used trace-driven simulation to evaluate the effects of instruction prefetching strategies. However, their proposal to exploit the structural locality of program instruction references involves modifications to the CPU architecture rather than the memory subsystem [Lee84]. Their analysis focused on methods of avoiding pipeline breaks when instruction branches are executed. They proposed a four-state branch predictor and showed through trace-driven simulation, that it could be expected to provide an improvement of 5 to 20 percent in CPU performance.

2.5.4 Proposed Architectures Tailored to Symbolic Workloads

As mentioned in Chapter 1, most architectures tailored for symbolic workloads are designed to exploit some high-level characteristic of this type of workload. This was most evident with the MIT Lisp Machine [Gree84] which was optimized for fast function calling, had a large virtual address space, and had a heavily-microcoded architecture to support high-level Lisp functions. As this section will show, another major characteristic of symbolic workloads exploited by system designers include symbolic workloads' heavy use of lists for their data structures.

Hayashi, Hattori, and Akimoto tailored their memory subsystem design to exploit the high temporal locality of variables that is due to the frequent function calling characteristic of symbolic workloads. In 1983, they published the results of their implementation of a high-speed hardware stack in a back-end Lisp processor to exploit this temporal locality of variables [Haya83]. The variable values are cached in the hardware stack and are tagged with the function entry frame number. This was predicted to speed up function calling for functions with deep-bound variables and double the overall performance of the system [Yuha86]. However, since their Lisp system, UTILISP, only used shallow binding of variables, the effect of this cache could not be measured. Because of the spatial locality of list cells, they implemented a garbage collection scheme using reference counts, but only storing the counts of references outside the subspace containing the cell. The excellent performance of this garbage collection scheme added further evidence to the static spatial locality of memory references.

Sohi, Davidson, and Patel have proposed an architecture specifically tailored for symbolic workloads in [Sohi85a]. This paper is based on Sohi's research which is fully described in [Sohi85b]. Sohi proposed representing each Lisp list as a tree in a logical address space with the *car* of the list cell being located at loca-

tion $2n$ and the *cdr* of the list cell being located at location $2n + 1$ where n is the location of the list cell. Thus the location of any element of the list can be computed arithmetically rather than by following a set of pointers. Since most lists are almost linear, Sohi proposes storing in the virtual address space only the leaf nodes to the tree which are the atoms in the list and other nodes that contained forwarding pointers due to *rplaca* and *rplacd* operations on the list or space limitations on the list. In the best case, this allows the list representation to be about four times as compact as the two-pointer list cell and twice as compact as lists represented using *cdr*-coding. Sohi further proposed that the machine architecture consist of separate processors for list processing and garbage collection and that the exception tables for lists currently being accessed be stored in a content addressable memory (CAM) cache. He also proposed a Tree Occupancy Table, analogous to a page table, to hold the starting locations of list exception tables in the CAM cache. To validate his proposed architecture, he demonstrated the greater efficiency of this architecture in implementing the higher level Lisp *length* and *member* functions.

Reese has also proposed an architecture tailored to symbolic workloads by more efficiently supporting those high-level Lisp functions that occur most frequently in Lisp programs such as *get* and *nthelem* [Rees85]. He constructs a Lisp architecture, ALISP, that allows constant time access to all top-level list elements by using a CAM to cache lists in the current working set of the program. He specifies an alternate list representation similar to that used for *cdr*-coding but with an additional tag field to specify the position of a list cell in the list. This additional tag then allows any top-level element of the list to be retrieved in constant time using associative search. The CAM controllers are designed to perform some high-level functions such as *reverse* independently of the CPU, thus allowing for some parallelism in the Lisp function execution. In addition, once the CPU has written a pointer to a CAM controller, the CAM controller is able to perform garbage collection on the list in its

cache in parallel with other CAM controllers and independently of the CPU. Reese validated his architecture by simulating the performance of a resolution theorem prover on his architecture and showing that it achieves an 80 percent speedup over clause hashing techniques on a conventional architecture and a 235 percent speedup over the conventional architecture when clause hashing is not used.

In [Thaz86], Thazhuthaveetil proposed an architecture tailored to symbolic workloads that relies on caching identifier-pointer pairings for all lists that are accessed. In addition, the identifier-pointer pairings of the *car* and *cdr* of a list are cached in a list processor table when a list is first accessed. A separate list processor manages the heap memory and maintains the list structure while the evaluation processor works strictly with the object identifiers in the cache. Because of the separate processors, the evaluation processor is often able to continue its execution while the list processor makes the appropriate changes to the list structure in heap memory. The list processor maintains a reference count for each line in the cache and reclaims both the cache line and the heap memory whenever a reference count goes to zero. Thus, the garbage collection is incremental and is overlapped with the evaluation processor's execution. Thazhuthaveetil simulated his proposed architecture using list structures corresponding to the study by Clark and Green [Clar77] and to his own studies of the effective branching factors in lists. He found that the list processor table captured the temporal locality of list accessing about as effectively as a standard LRU cache but was inferior to the LRU cache in exploiting the spatial locality of the list structure since only the *car* and *cdr* of a list were cached. However, since the list processor table permitted overlapping of the execution of the evaluation processor and list processor and more efficient garbage collection, the architecture was deemed more effective for symbolic workloads.

Finally, Patterson and his research group have tailored the Berkeley RISC-

II architecture for symbolic workloads [Tayl86]. This SPUR Lisp architecture differs from the RISC-II architecture in only a small number of features. It has four new compare and branch conditions which depend on the tag values of the operands and hardware checking for five kinds of tag exceptions during data operations. With these few modifications, they predict, based on their simulations, that the SPUR will run Common Lisp programs at least as fast as the Symbolics 3600 Lisp machine or a DEC VAX 8600.

2.6 Contributions of This Research

This research is the first systematic characterization of the spatial, temporal, and structural locality of symbolic workloads based on the word-level analysis of virtual memory address traces. As such, it lays a foundation for future work in this area by providing a methodology for a comprehensive analysis of program locality. This methodology includes new metrics for spatial, temporal, and structural locality as well as adapting a semi-Markov model used in Lewis and Shedler's page fault studies for use in characterizing the word-level virtual memory addressing behavior of both symbolic and conventional workloads. These new metrics are then used in conjunction with older metrics mentioned in the previous sections to identify the special locality characteristics of symbolic workloads. Finally, this research evaluates a new memory subsystem design tailored for symbolic workloads by exploiting these special locality characteristics and exercising new design alternatives. Thus, this research adds to the published work described in each of the previous sections of this chapter.

2.7 Summary

Research into program locality has taken many different approaches from direct analysis of trace data to inferences of program locality based upon trace-

driven simulation and from high-level program analysis to low-level memory word referencing behavior. Unfortunately, as is readily apparent from this chapter, there has been little continuity in the research directions taken, nor in integrating the results of the various approaches to produce a picture of program referencing behavior consistent with the many varied results. As will be seen in subsequent chapters, this research incorporates previous results, where possible, and integrates the new locality characteristics revealed by this research with the previous results of this chapter to provide explanations for the measured differences in the virtual memory addressing behavior of symbolic and conventional workloads.

Chapter 3

Data Collection

This chapter describes the methodology used to collect the virtual address traces used for this research. It begins with a description of the overall experimental setup and then describes, in detail, the modifications made to the Explorer II microcode to enable recording of the virtual memory address references and machine control register information. Next, the procedure for executing the workloads and collecting the trace data is detailed. The selection criteria and modifications made to each workload are next described, and the chapter ends with a description of the methodology used to extract samples from the workload address traces.

3.1 Overview of the Experimental Setup

This research consisted primarily of the development and use of the seven modules shown in Figure 3.1. The trace program was a Lisp program that started the execution of the program being traced while it was also activating the microcode modifications in the TI Explorer II that record the address traces. These traces were stored on a 16 Mbyte memory board whose physical address space was not mapped into the virtual memory address space and so was inaccessible to the operating system for virtual memory reads and writes. Another Lisp program then wrote the trace data to a disk file for input to the sort/data compression routines. Sorting and data compression reduced the amount of data to the minimum required for adequate characterization of the memory referencing behavior. Then the data analysis routines

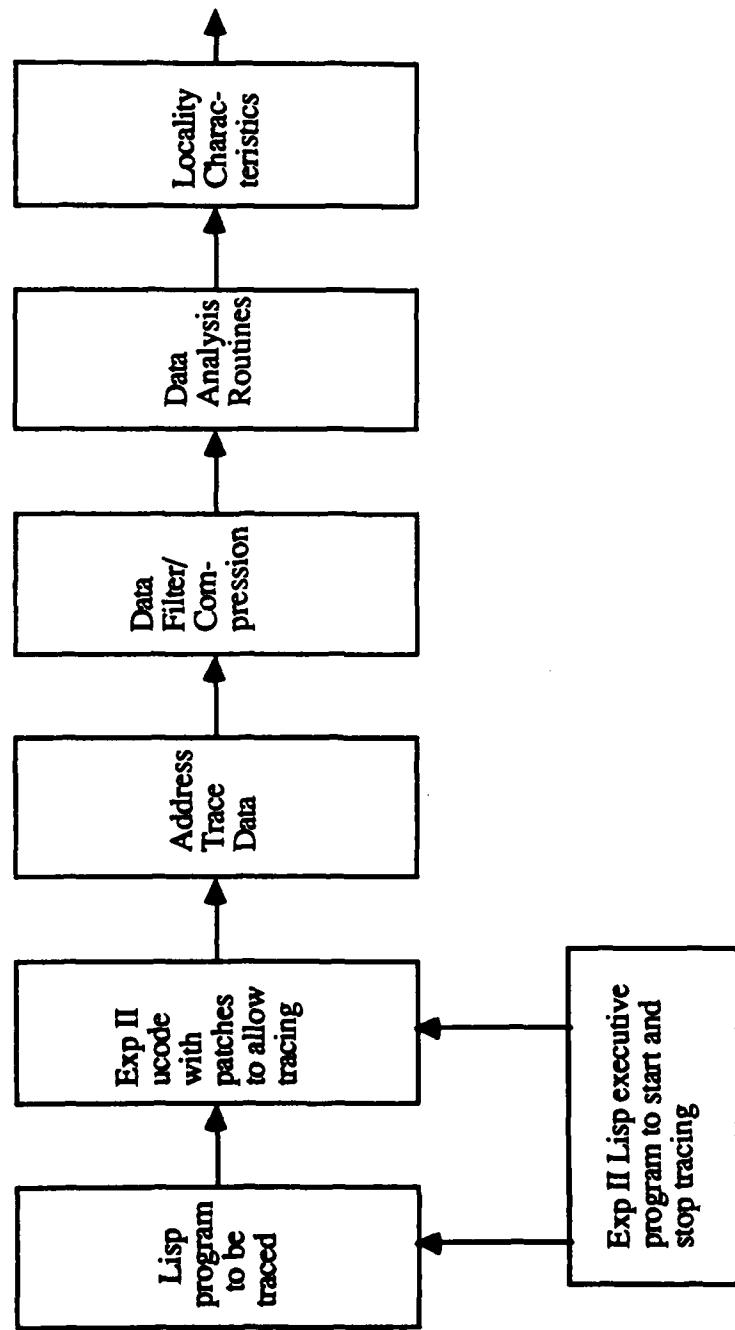


Figure 3.1: Project Overview

computed the locality characteristics of the memory accesses which were, in turn, used to optimize the memory subsystem design for this workload.

3.2 Microcode Modification

With considerable assistance from the microcode programmers and processor board designers at Texas Instruments, I modified the microcode on the Explorer II to record address traces in a manner similar to that of Agarwal, Sites, and Horowitz [Agar86]. However, rather than modifying each microcode location that could generate a memory reference, the page map table was modified to force the system on every virtual memory reference to enter the page fault abort handler, which was modified to record the trace data. Along with the virtual address, the following information was also recorded: whether the reference was a read or a write, whether an instruction or data word was being referenced, whether a data word was being prefetched, and whether a reference had been made to a forwarding pointer. This information was recorded on a memory board whose configuration ROM had been removed causing it to fail the power up check and thus be excluded from the address space of the system. However, reads and writes could be made to the board using physical addresses, and the microcode in the page fault abort handler used this procedure to record the virtual memory address trace information. After the trace information had been recorded, the modified microcode then used the other information bits in the page map table to restore proper access to the memory location and completed the memory reference. Finally, the microcode reset the page map table access bit to zero to insure that a page fault would occur again when this page was next referenced. This scheme had a distinct advantage over the approach used by Agarwal, Sites, and Horowitz, since the microcode that actually recorded the trace data was localized in the page fault abort handler and did not have to be replicated throughout the microcode. This allowed for easier modification of the code and also

simplified the debugging of the modified microcode.

A Lisp program was also written to set in the microcode a bit that reset all the access bits in the page map table and enabled the other microcode patches to be executed. At the end of the program trace or when the memory board was filled with trace data, the microcode restored the page map table to its normal state and disabled the other microcode patches.

3.3 Trace Collection Procedure

Because my goal was to characterize symbolic workload behavior and not overall system behavior, I traced the programs with all interrupts disabled and the incremental garbage collector turned off. Thus, my research established a baseline for symbolic workload behavior which subsequent research can use to characterize the effects of context switching and incremental garbage collection on the word-level virtual memory referencing behavior. I also traced that portion of the workload that did not involve extensive amounts of disk or screen I/O since the memory referencing behavior for these activities was heavily influenced by the TI Explorer II paging algorithms and display routines.

The first step in the trace procedure was to cold boot the Explorer II under the modified microcode and with the configuration ROM removed from the 16 Mbyte memory board. Then the compiled code for the workload to be traced was loaded and executed immediately after setting the microcode trace-enable variable to *true*. The trace collection was stopped when either the 16 Mbyte board was full or when the workload being traced terminated—whichever occurred first. As stated in Section 2.5.2, because the modifications were made to the microcode, the execution time of the workload was only increased by a factor of 10, rather than the typical slowdown of 100 to 1000 typical of simulators or operating system trace modes.

After the trace was completed, a function profiling routine dumped to a file the addresses in trace memory of the instruction fetches, the virtual memory address which was the target of the instruction fetch, and the name of the functions in which the fetches took place. To decrease the size of this file, only those instruction fetches which occurred in a function different from that of the previous instruction fetch were recorded. Thus, by using this function profiler, the major phases that each workload went through as it executed, and the section of trace memory corresponding to each of these phases was determined.

All of the trace information was then converted from its binary-encoded form to an ASCII hexadecimal representation and dumped from the 16 Mbyte board to a file on disk. This file was then used as the source file for the data sampling routine. Once these tasks were complete, the Explorer II was powered off, the configuration ROM replaced, and the system would be cold booted under an unmodified release of the system microcode.

3.4 Workload Selection

3.4.1 Selection Criteria

There is no universally accepted distinction between symbolic and conventional workloads. Many, in fact, question such a categorization of all workloads into just these two categories. However, since over the past decade there has been an extensive effort to tailor computer architectures and implementations to the symbolic processing tasks required for AI applications, categorizing workloads in this manner at least provides the framework for analysis of the distinctive features present in AI applications.

Two principal factors were considered in selecting the symbolic workloads to be traced. Workloads were selected that represented legitimate AI applications

Table 3.1: Explorer II Workloads Traced

<u>Workload Name</u>	<u>Application</u>	<u>Categorization</u>
BIASLisp	Circuit Analysis	Conventional
Boyer	Theorem Prover	Symbolic
Compile Buffer	Lisp Compiler	Symbolic
FFT	Numeric Computation	Conventional
GLISP	Expert System Tool	Symbolic
QSIM	Qualitative Reasoning	Symbolic
Reducer	Symbolic Computation	Symbolic
TMYCIN	Expert System Tool	Symbolic

such as qualitative reasoning, theorem proving, and expert systems reasoning, and not just conventional applications written in Lisp. But, compilers as well as symbolic computation programs such as Macsyma were also included in the symbolic workload category because of their inherent symbolic processing tasks and their data-driven behavior. The other criterion was more pragmatic: preference was given to applications that were portable to the TI Explorer without unreasonable effort.

For conventional workloads, Lisp programs were selected that implemented conventional applications such as numeric computation and circuit analysis. Here, preference was given to those workloads that made minimal use of list structures, and instead relied primarily on arrays, hash tables, and records. Table 3.1 lists the workloads which were traced.

3.4.2 Modifications Made to the Workloads

Each workload was kept as close to its original form as possible with only those modifications made that were necessary to allow the program to run on the Explorer II or to allow the program to run without input from the keyboard. Without these latter modifications, the process would halt when input was required from the keyboard, and the trace memory would be filled during the execution of one of the operating system processes. Some modifications were also made to minimize the effects of print statements. Specifically, screen print statements were commented out where possible or, at the least, redirected to a null stream rather than to the terminal handler routines. Finally, the call to the test-trace macro was sometimes embedded in a workload subroutine that was executed after the initial workload housekeeping functions, to delay the tracing until the nucleus of the program was actually executing. For example, when the qualitative reasoning system was traced, the tracing was delayed until the user menu selections had been made and the actual qualitative simulation of the selected system had started.

3.5 Extraction of the Sample Data from the Workload Traces

3.5.1 Determination of the Sample Length

A sample length of 450,000 references was selected based upon both a high-level analysis of the workload phases and a low-level analysis of the sensitivities of the locality characteristics to the trace sample length. From the high-level analysis, a length of 450,000 references was found to be an upper bound on the length of the trace since trace samples longer than this would sometimes contain more than one workload phase, thus masking the characteristics of a particular phase. From the low-level sensitivity analyses, it was found that the locality characteristics were very similar despite variations in the trace sample length from 10,000 to 450,000 memory

references. This is typified by the plots of the cumulative distribution of spatial distances for the TMYCIN expert system overall reference strings, shown in Figures 3.2 through 3.4. Thus, the 450,000 reference sample length was small enough to allow analysis of individual workload phases, but large enough to provide enough references, even after sorting into subtraces¹ to allow accurate computation of the locality characteristics.

3.5.2 Extraction of the 450,000 Reference Sample

The function profiler described in the previous section was used to extract 450,000 reference segments from the address trace file for each major phase of each workload. All but two workloads had only one major phase. These two, the GLISP expert system tool and the Explorer Lisp compiler, each showed two distinct phases making them, in essence, four workloads altogether. So, for these two workloads, a 450,000 reference segment was extracted for each of the two phases.

3.6 Summary

This chapter summarizes the procedures used to produce the virtual memory address traces upon which this research is based. Thus, it allows a qualitative evaluation of the validity of the virtual memory address traces and so of the research overall. Wherever possible, the experimental setup was designed to produce virtual memory address trace samples that were representative of the memory referencing behavior of the types of workload being characterized. This objective guided each step of the data collection process, from the design of the microcode modification to the extraction of the virtual memory trace sample for analysis.

¹The sorting criteria are discussed in the next chapter.

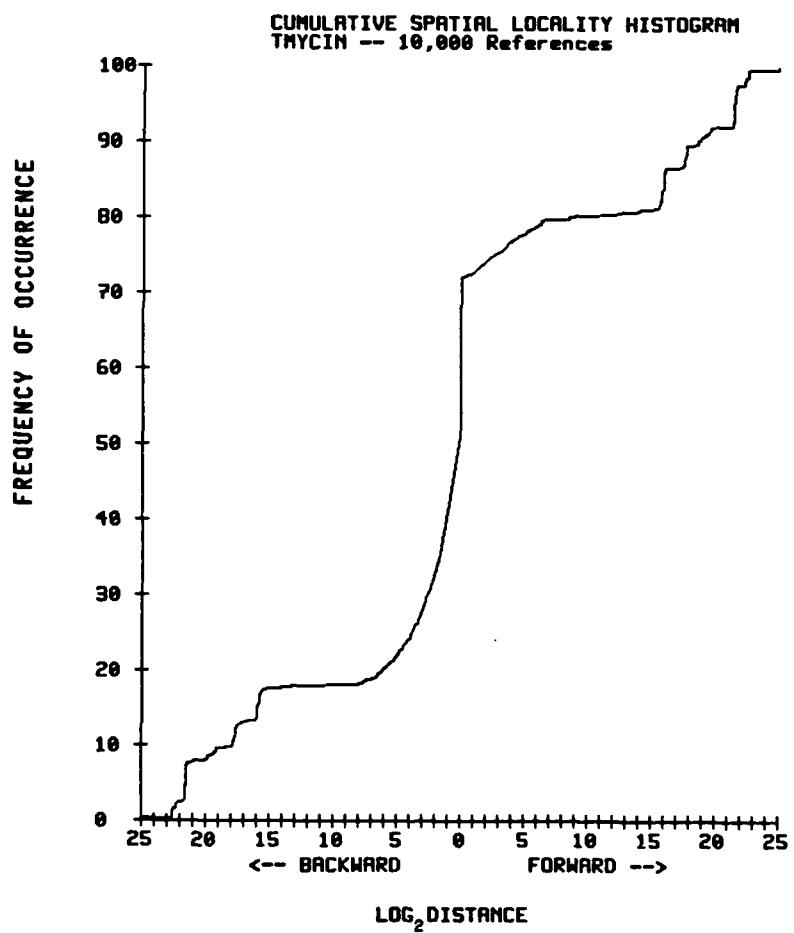


Figure 3.2: Example Spatial Locality Histogram for a 10,000 Reference TMYCIN Sample

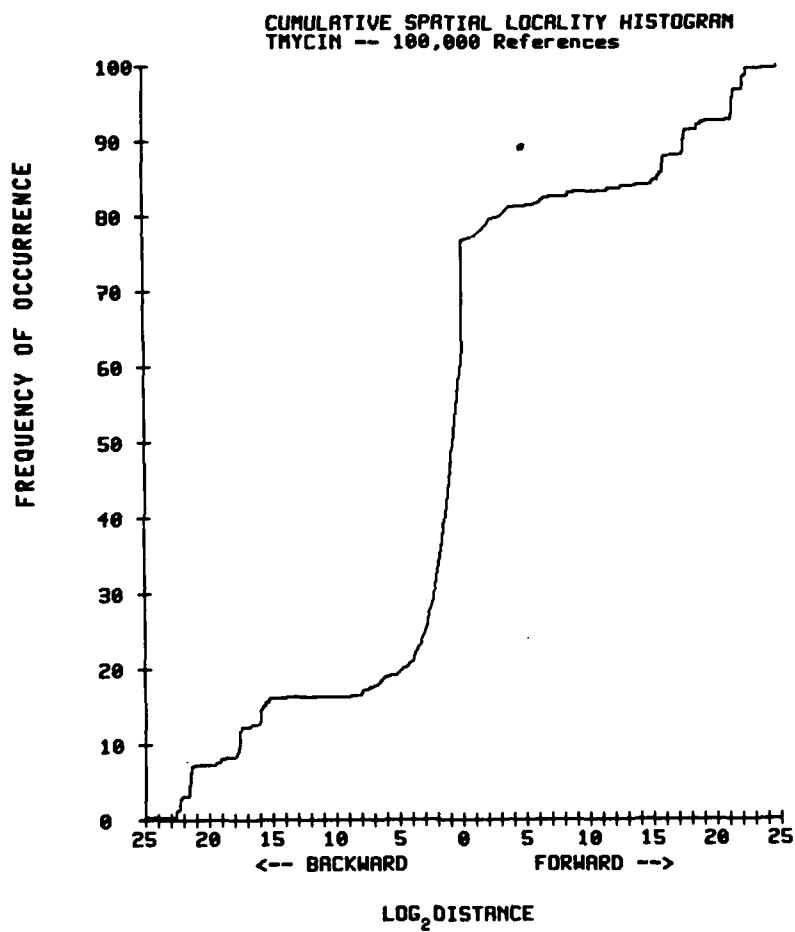


Figure 3.3: Example Spatial Locality Histogram for a 100,000 Reference TMYCIN Sample

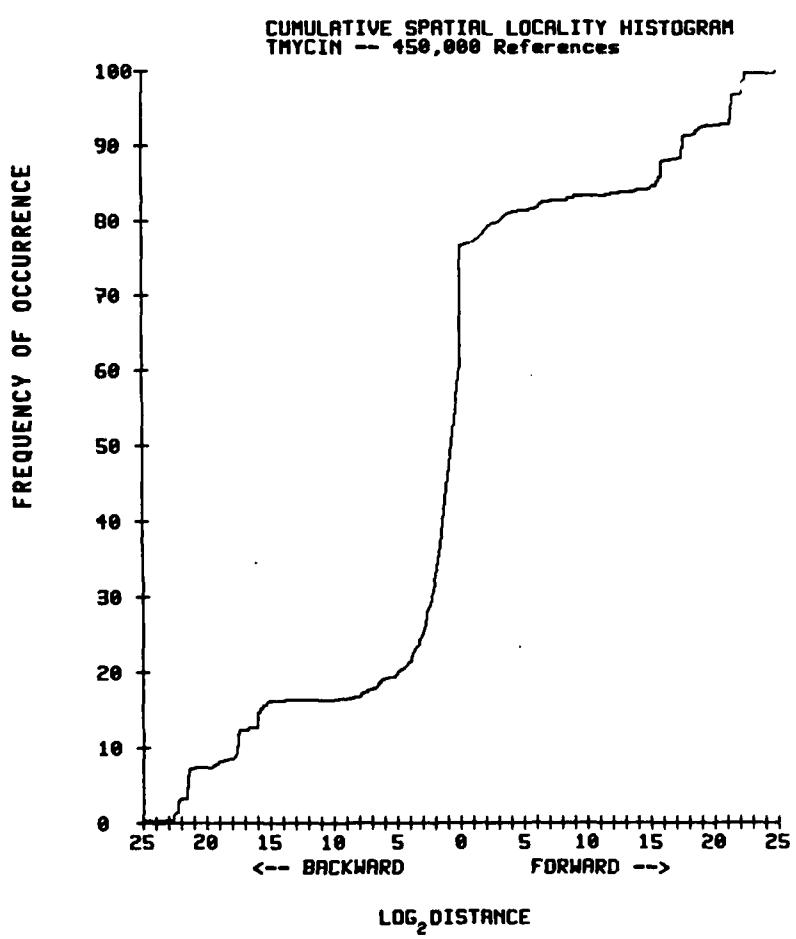


Figure 3.4: Example Spatial Locality Histogram for a 450,000 Reference TMYCIN Sample

Chapter 4

Trace Analysis

This chapter describes the methodology for analyzing the 450,000 reference samples. Since much of this methodology is based upon a new model of word-level memory referencing behavior, this model is first described and the motivation for it explained. Although most of the trace analysis results are described in the following chapter, those results which helped to validate this model of program behavior are discussed here. The chapter then describes the sorting techniques used. Finally, each of the locality characteristics computed is described as well as the usefulness of each in characterizing memory referencing behavior.

4.1 Markov Model of Low-Level Memory Referencing Behavior

It was obvious features of the temporal distance strings themselves that motivated this model of low-level memory referencing program behavior. Previously unreferenced virtual memory locations, which will be called 'new references', occurred most often in clusters. Strings of references to previously referenced memory locations, which will be called 'old references', would be followed by periods of successive new references in a behavior analogous to that noted by Denning and others in page fault studies [Denn68]. Much more surprising, at first glance, was a tendency in the temporal distance string for a large number of successive references to have the same stack distance. Adopting Thazhuthaveetil's term, this phenomenon has been attributed to the structural locality of the workload, since the memory contents are

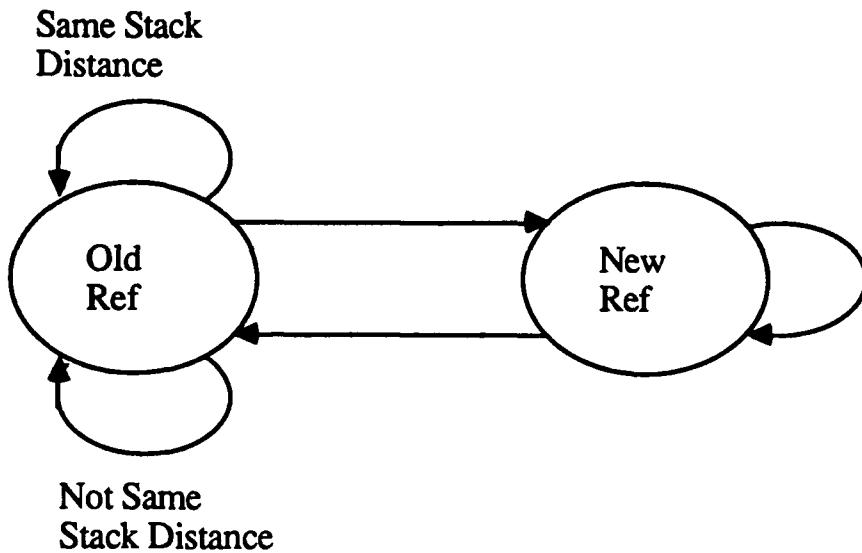


Figure 4.1: Two-State Markov Model of Program Behavior

being referenced in the same order in which they had been previously referenced [Thaz86].

To characterize the types of behavior observed in the temporal distance strings, a two-state Markov model shown in Figure 4.1 and similar to that used in models of paging behavior was developed [Lewi73]. The program referencing behavior transitions between long periods of old references—the Old-Ref state—followed by bursts of new memory references—the New-Ref state. Within the Old-Ref state successive references with the same stack distance—Same-Stack-Distance or SSD references—are distinguished from successive references with different stack distances—Not-Same-Stack-Distances or NSSD references. Altogether, then, there were five possible state transitions: New-New, New-Old, Old-New, Old-SSD, and Old-NSSD.

A three-state Markov model was also considered to characterize the symbolic workload behavior by separating the Old-Ref state into a Same-Stack-Distance

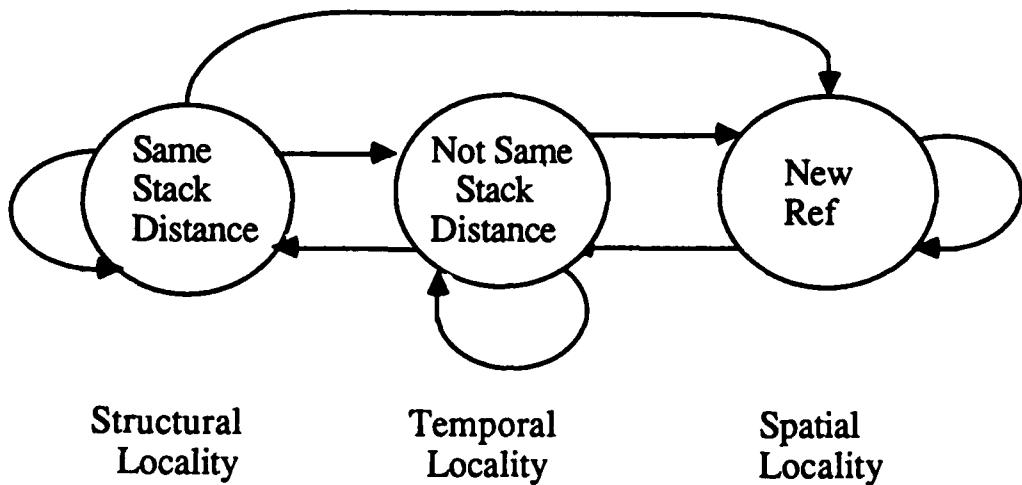


Figure 4.2: Three-State Markov Model of Program Behavior

state and a Not-Same-Stack-Distance state to gain additional insight into the memory referencing behavior when structural locality is present. This model is shown in Figure 4.2. However, because this model has almost twice as many transitions and provides little additional information, the three-state model was rejected in favor of the simpler and more tractable two-state model.

When the distance strings were computed for just those references in each transition class, the strings differed markedly from each other and revealed characteristics not evident in the overall distance string. In addition, many of the transition distance strings showed a strong degree of order, and therefore predictability, that was not present in the overall distance string. This along with other results presented in Chapter 5 validated the model's usefulness in providing a deeper understanding of the memory referencing behavior.

Table 4.1: Relative Frequency of Types of References

Type Ref	# of Refs		% of Total		# of Addresses		% of Total	
	Sym	Conv	Sym	Conv	Sym	Conv	Sym	Conv
All	450,000	450,000	100	100	15,258	35,344	100	100
Inst	173,161	57,000	38.5	12.7	2,861	379	18.8	1.1
Data	276,839	393,000	61.5	87.3	12,399	34,965	71.2	98.9
D Read	246,307	281,120	54.7	62.4	10,805	33,628	70.8	95.1
D Write	30,532	111,880	6.8	24.9	8,487	31,229	55.6	88.4

4.2 Trace Categorization and Data Compression

The address traces were sorted into five types of address references: all references, instruction fetches, all data references, data reads, and data writes. Then each of these resultant traces was further sorted into four of the five transition types: Old-New, Old-NSSD, New-Old, and New-New. Table 4.1 gives the mean number of each type of reference for symbolic and conventional workloads as well as the number of distinct virtual memory addresses referenced by each type of reference. As shown in Table 4.1, the mean number of instructions in the symbolic workloads was about three times the mean number of instructions in the conventional workloads. Also, the mean number of data writes for the symbolic workloads was less than one-third that of the conventional workloads. These differences in the relative frequency of the types of references for symbolic and conventional workloads are used in Chapter 5 to account for some of the measured differences in symbolic and conventional workload locality.

To compress the data, distance strings rather than the actual memory reference addresses were analyzed. While the spatial distance string was simply com-

puted by subtracting the virtual memory address from the previous virtual memory address, the temporal distance string computation was more involved. The temporal distance was computed by simulating an LRU stack and by recording the stack position of each simulated LRU stack access. Zero was used to represent a new reference to the stack and a Lisp implementation of Spirn's temporal distance computation algorithm was used to compute this string [Spir77].

Both spatial and temporal distance strings were computed for the full set of addresses for each type of address reference. In addition, using the traces sorted by transition type, the following distance strings were also computed: an Old-New spatial distance string, an Old-NSSD temporal distance string, a New-Old temporal distance string, and a New-New spatial distance string. Thus, altogether, 30 distance strings were computed for each workload phase as shown below:

(5 Ref Types) x [(3 Spatial Loc) + (3 Temporal Loc)]

All Refs

Inst Fetches	All Refs	All Refs
All Data Refs	New-New	Old-NSSD
Data Reads	Old-New	New-Old
Data Writes		

4.3 Locality Characteristics Computed

Four sets of locality characteristics were computed for each of the distance strings, making a total of 120 sets of locality characteristics for each workload phase. These four locality characteristics for each distance string were an individual frequency histogram, a cumulative histogram, a correlogram, and a power spectrum.

The individual frequency histogram, analogous to a probability density

function, mapped the range of possible distances to the x-axis and the percent of the total number of references that had each distance to the y-axis. Since the spatial distance space was very large extending from 2^{25} words backward to 2^{25} words forward, these distances were mapped to the x-axis logarithmically with the spatial distances of -1, 0, and 1 being mapped to 2^0 . The temporal distance space extending from 1 to the maximum LRU stack depth referenced, on the other hand, was smaller and so was mapped linearly to the x-axis. Using these individual frequency histograms, typified by Figure 4.3, it was possible to identify those distances that occurred most frequently.

The cumulative histogram, analogous to a cumulative distribution function, used the same mappings for the x-axis described for the individual frequency histogram. However, each y-axis value, corresponded to the percent of the total number of references that had a distance equal to or to the left of the corresponding value on the x-axis. Therefore, the cumulative histograms showed the overall distribution of the spatial distances over the virtual memory address space and the distribution of the stack access positions over the simulated LRU stack. They were particularly helpful in identifying the ranges of distances corresponding to large percentages of the total number of references. Figure 4.4 illustrates this locality characteristic.

The correlogram, a plot of a distance string's autocovariance coefficient for each lag from 0 to 120, revealed a number of characteristics including the rate at which the autocovariance of the distance string dropped off with the lag, the magnitude and duration of periodicities in the distance string, and how the magnitudes of these periodicities varied with the lag. Figure 4.5, an example correlogram, shows, for this particular distance string, rapid fluctuations in autocovariance with lag but centered around a constant value.

The power spectrum of the distance string, computed by taking the Fourier

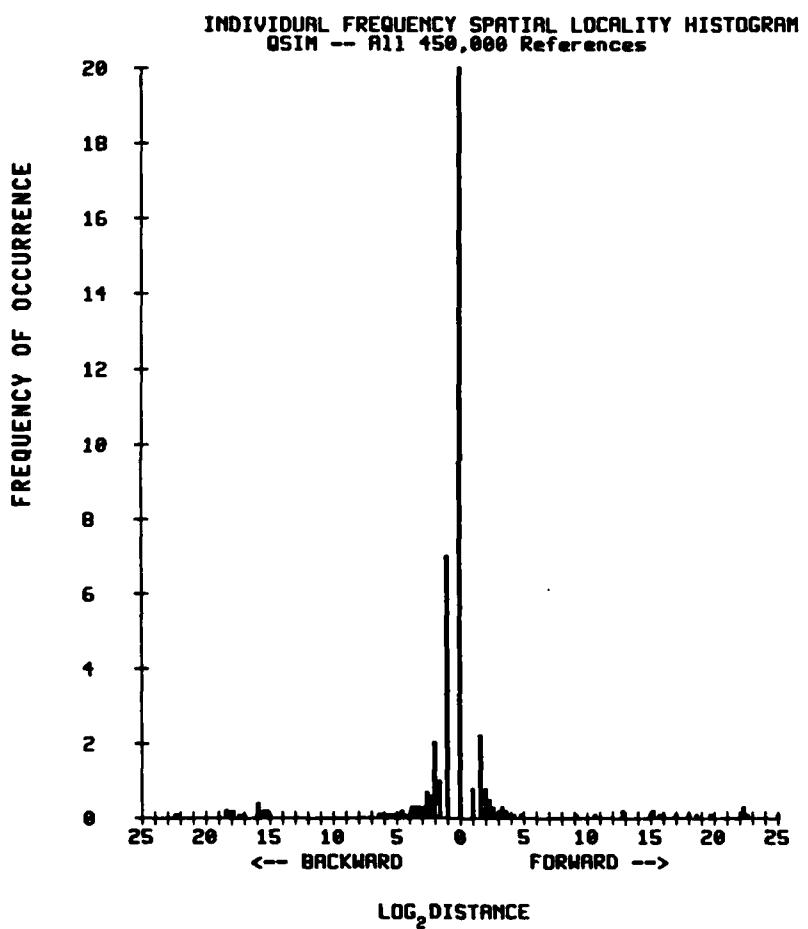


Figure 4.3: Example Individual Frequency Histogram

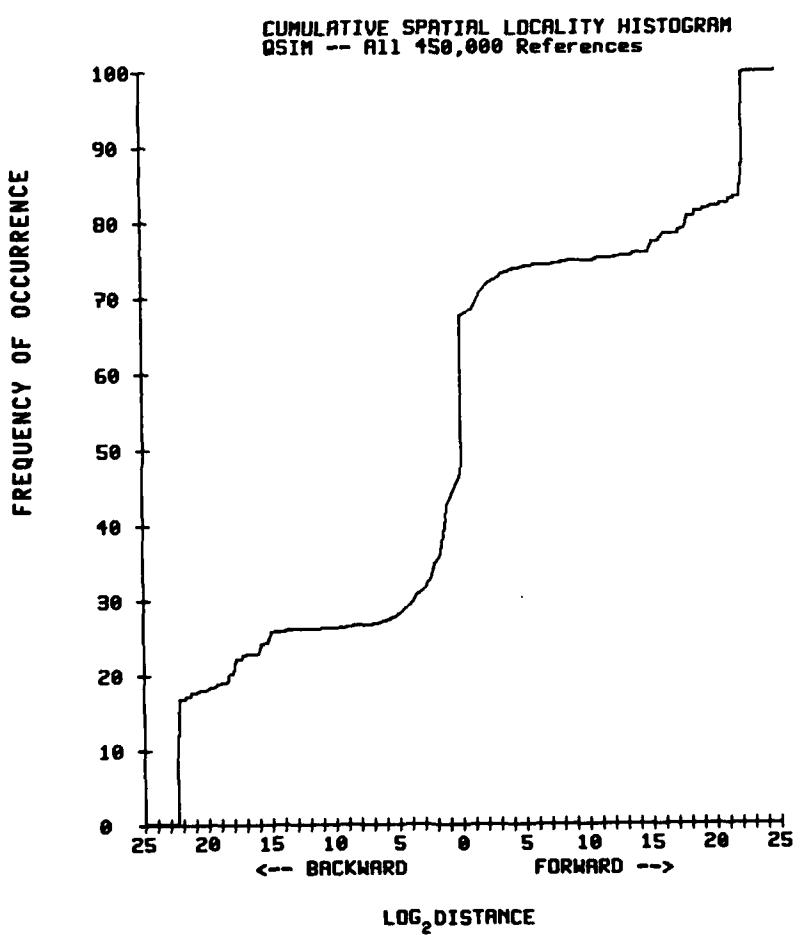


Figure 4.4: Example Cumulative Frequency Histogram

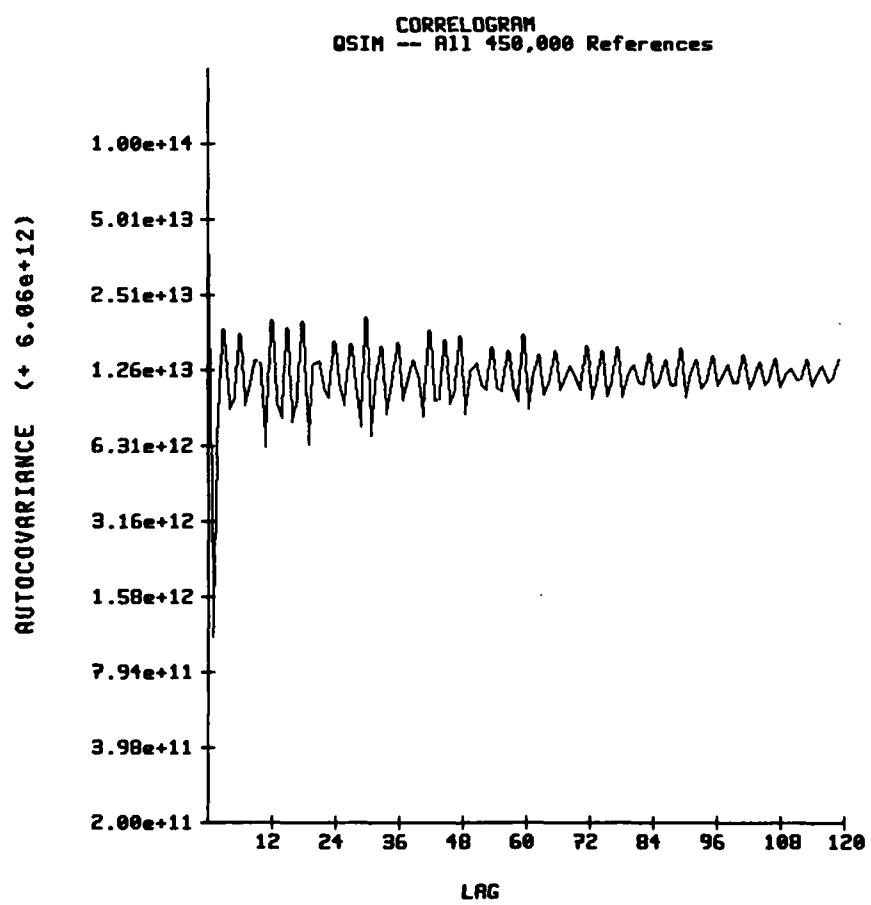


Figure 4.5: Example Correlogram

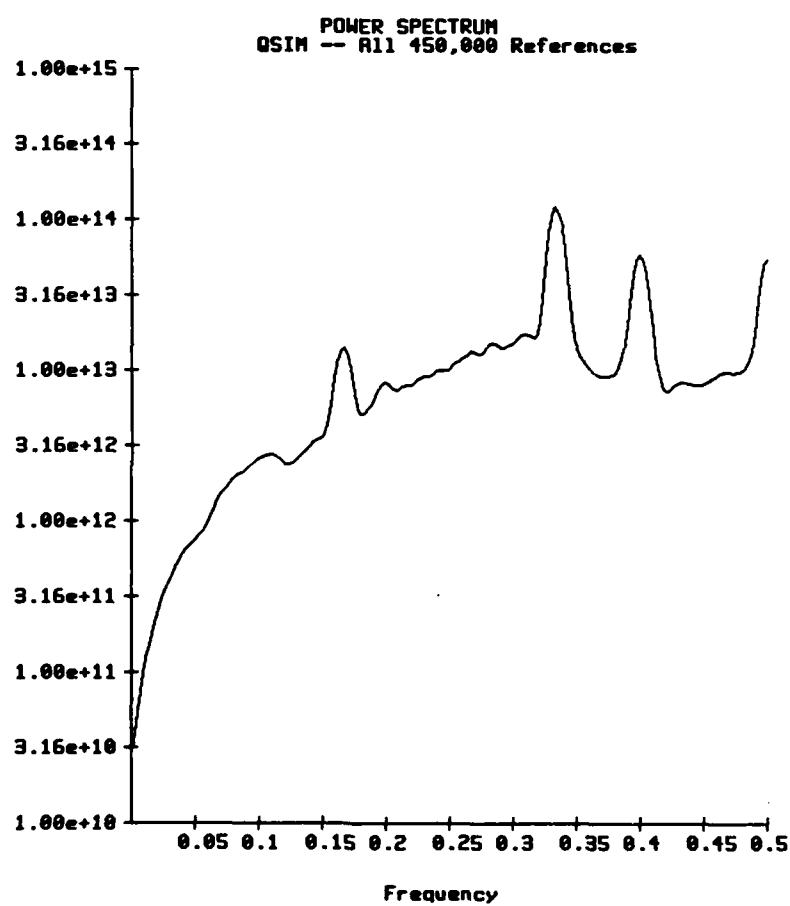


Figure 4.6: Example Power Spectrum

transform of the autocovariance function, more clearly revealed the exact values of the periodicities and whether the distance string was correlated more strongly over very short sequences (less than five references in length) or over longer sequences. As shown in Figure 4.6, the rapid variations in autocovariance in Figure 4.5 correspond to values of 0.5, 0.33, and 0.17 reflecting periodicities in the distance string of 2, 3, and 6 respectively. There is also a peak at 0.4 corresponding to an average periodicity of 2.5. Finally, Figure 4.6 shows that the distance string is correlated more strongly over short lags than over longer lags.

4.4 Summary

Thus, through systematically tracing both symbolic and conventional workloads and then computing these characteristics, a comprehensive data base was produced for analyzing the differences and similarities of these two types of workloads. Computing both 'binning' characteristics, the histograms, as well as 'order' characteristics, the correlograms and power spectra, enabled this research to view both the 'order' and 'binning' aspects of the memory referencing behavior.

Chapter 5

Results

This chapter describes the measured differences in the spatial, temporal, and structural locality characteristics of symbolic and conventional workload word-level virtual memory referencing behavior. It also introduces new measures of memory referencing locality used to characterize the behavior of the workloads and ends by describing a difference in the transition probabilities for the two-state Markov model between symbolic and conventional workloads. This chapter fully describes each of the noted differences between symbolic and conventional workloads and gives a statistical measure of confidence for each difference¹. Other known characteristics and measurements of symbolic and conventional program behavior are then used, where possible, to explain the differences. Finally, the implications of each difference for memory subsystem design are addressed.

5.1 Spatial Locality

5.1.1 Existence of a Spatial Locality Window

The cumulative histograms of the spatial distance strings revealed a heretofore unpublished characteristic of word-level virtual memory referencing behavior. In all workloads and for all five types of address references, if spatial locality existed in the virtual address space, it existed within a window of plus or minus 32 words from

¹The detailed data for all Explorer II workloads from which the results were computed are included as Appendix A.

the current address. Very few subsequent references took place between 32 words and 32K words from the current reference for this 32-Mword address space. This is shown in Figure 4.4 by the almost zero slope segments of the cumulative histogram curve between these values.

These plateaus in the cumulative spatial histogram curve can be better understood by viewing the virtual address space as several subspaces including: a subspace for system tables in very low memory, a subspace for system Lisp routines about 4 Mwords higher in the virtual address space, and a user subspace about 4 Mwords higher still. At load time, memory is allocated in the user subspace for the compiled code and any static structures such as arrays. Also at load time, any system functions called in the user code are linked to the appropriate function headers in the lower virtual memory address subspaces. During execution of the program, additional heap memory is dynamically allocated from the user subspace to construct new list elements and other dynamic structures. With this understanding, then, references falling within the spatial locality window represent subsequent references to the same subspace, while references outside the window represent reference transitions between subspaces or references to different functions within the system Lisp routine subspace. The narrowness of the window shows that references within the user subspace have very high spatial locality.

To further characterize this phenomenon of a narrow spatial locality window, a new spatial locality metric was introduced. This metric, the spatial window probability (P_{SW}), is defined as follows:

P_{SW} : the probability that the subsequent memory reference is within plus or minus 32 words of the previous reference.

Thus, P_{SW} is basically the difference in the values of the upper and lower plateaus in the cumulative spatial locality histograms. Using this metric, the first significant

Table 5.1: Spatial Window Probability (P_{SW})

Type Ref	Symbolic		Conventional		Ratios		t-Test Used	Conf Level
	Mean	SD	Mean	SD	Mean	F		
All	0.494	0.0587	0.332	0.0255	1.488	5.32	Eql	0.006
Inst	0.907	0.0407	0.957	0.0594	0.948	2.13	Eql	0.187
Data Refs	0.438	0.1397	0.329	0.0106	1.334	173	Eql	0.319
Data Reads	0.406	0.1531	0.296	0.0120	1.378	162	Eql	0.357
Data Writes	0.504	0.1791	0.202	0.0396	2.496	20.5	Eql	0.053

difference between symbolic and conventional workload memory referencing behavior can be characterized.

5.1.2 Differences in P_{SW} for Symbolic and Conventional Workloads

From the individual P_{SW} values for each workload, a mean P_{SW} was computed for the symbolic workloads and another mean P_{SW} for the conventional workloads. Table 5.1 summarizes the differences in spatial locality for the symbolic and conventional workloads by giving the means and standard deviations (SD) of the P_{SW} values for each type of workload. This table also gives the ratio of the mean symbolic P_{SW} to the mean conventional P_{SW} along with the F statistic used in the test for significantly different variances. This F statistic is computed as the ratio of the variances of the distributions of symbolic and conventional P_{SW} values with the larger variance always appearing in the numerator. If this F statistic indicates that the variances of the distributions of symbolic and conventional workload P_{SW} values are significantly different, then the Student's t-Test for Unequal Variances is used. Otherwise, the standard Student's t-Test is used. The Student's t-test used is indicated in Table 5.1 by Uneql and Eql respectively. Finally, the last column of

Table 5.1, the confidence level, contains the likelihood that the symbolic and conventional workload P_{SW} values came from distributions with the same mean. When the confidence level is less than 0.05, the differences between the symbolic and conventional workloads locality characteristics are considered to be statistically significant.

Thus, Table 5.1 shows that the spatial window probability differs significantly for symbolic and conventional workloads for the memory reference stream consisting of all references. The mean P_{SW} for the symbolic workloads is almost 50 percent greater than that of the conventional workloads. And, using the equal variance Student's t-test for significantly different means, one can conclude that the probability that the values of P_{SW} for the symbolic workloads and the values of P_{SW} for the conventional workloads came from distributions with the same mean is only 0.006.

This difference does not show up when only instruction references are analyzed. In fact, as can be seen from Table 5.1, all workloads, both symbolic and conventional, have a very high P_{SW} for just the instruction fetches. Table 5.1 also shows that the difference in the spatial locality of symbolic and conventional workloads is not statistically significant when just data references are analyzed. Likewise, the differences in P_{SW} between the symbolic and conventional workloads are not statistically significant when data reads and data writes are analyzed separately, even though the mean P_{SW} for the data write spatial distance strings of the symbolic workloads is about two and one-half times that of the conventional workloads.

The difference in the spatial locality of the overall memory reference stream, (i.e., all memory references) is due to the much higher percentage of instruction fetches (with, as just mentioned, their higher spatial locality than data references) in the overall symbolic workload memory reference streams as noted in Chapter 4, and as shown in Table 4.1. Furthermore, since the symbolic workloads

consisted of more and smaller functions, there was a greater tendency for a number of instruction fetches to follow one another in the overall reference stream than was the case for the conventional workloads, which had much less function calling. Thus, the symbolic workloads experienced fewer transitions between the virtual memory subspaces.

5.1.3 Implications for Symbolic Memory Subsystem Design

These spatial locality results for the overall memory reference stream can be exploited in two principal ways. First, because the spatial locality window is very narrow, the size of prefetch blocks can be kept small. This reduces the bus bandwidth wasted by caching data² which are never referenced as well as the proportion of the cache taken up by these unneeded fetches. Second, there is no evidence that instruction fetches or data reads are any more or less spatially local for symbolic workloads than they are for conventional workloads. Thus, the same prefetching strategies used for conventional workloads should be used for symbolic workloads. However, the higher percentage of instruction fetches in the symbolic workloads may make prefetching more effective.

5.2 Temporal Locality

5.2.1 LRU Stack Distance Thresholds

Symbolic and conventional workloads also differ in their temporal locality. To compare these localities, a simulated LRU stack of the virtual memory addresses accessed was maintained for the entire memory reference string as described in the previous chapter. The metrics used to compare the symbolic and conventional work-

²The term 'data' used in this context refers to any contents of a memory location, i.e., either an instruction or data element.

loads are the stack distances corresponding to the 90, 95, and 99 percent thresholds in the cumulative temporal locality histograms and are termed LRU_{90} , LRU_{95} , and LRU_{99} respectively. Thus, these metrics are defined as follows:

LRU_{90} : the simulated fully-associative LRU stack depth required to capture 90 percent of the old references

LRU_{95} : the simulated fully-associative LRU stack depth required to capture 95 percent of the old references

LRU_{99} : the simulated fully-associative LRU stack depth required to capture 99 percent of the old references

These cumulative stack distance histograms are very close to the LRU hit function $h(m)$ defined by Wong and Morris [Wong88]. However, my cumulative temporal locality histograms only include old references, whereas their $h(m)$ LRU hit function includes both old and new references. Thus, whereas $h(m)$ corresponds directly to the hit ratio of a fully-associative LRU stack, my stack distance metrics correspond to the fully-associative LRU stack sizes required to capture 90, 95, and 99 percent of the old references, respectively. In addition, my metrics, as does $h(m)$, provide a lower bound measure on the interreference times between accesses to the same memory locations.

5.2.2 Stack Distance Thresholds for Symbolic and Conventional Workloads

As can be seen in Figures 5.1 and 5.2, between five and ten percent of the old references for the conventional workloads had very large stack distances of 25,000 or more. In contrast, none of the old references for the symbolic workloads had comparable stack distances as shown in Figures 5.3 through 5.10. Accordingly,

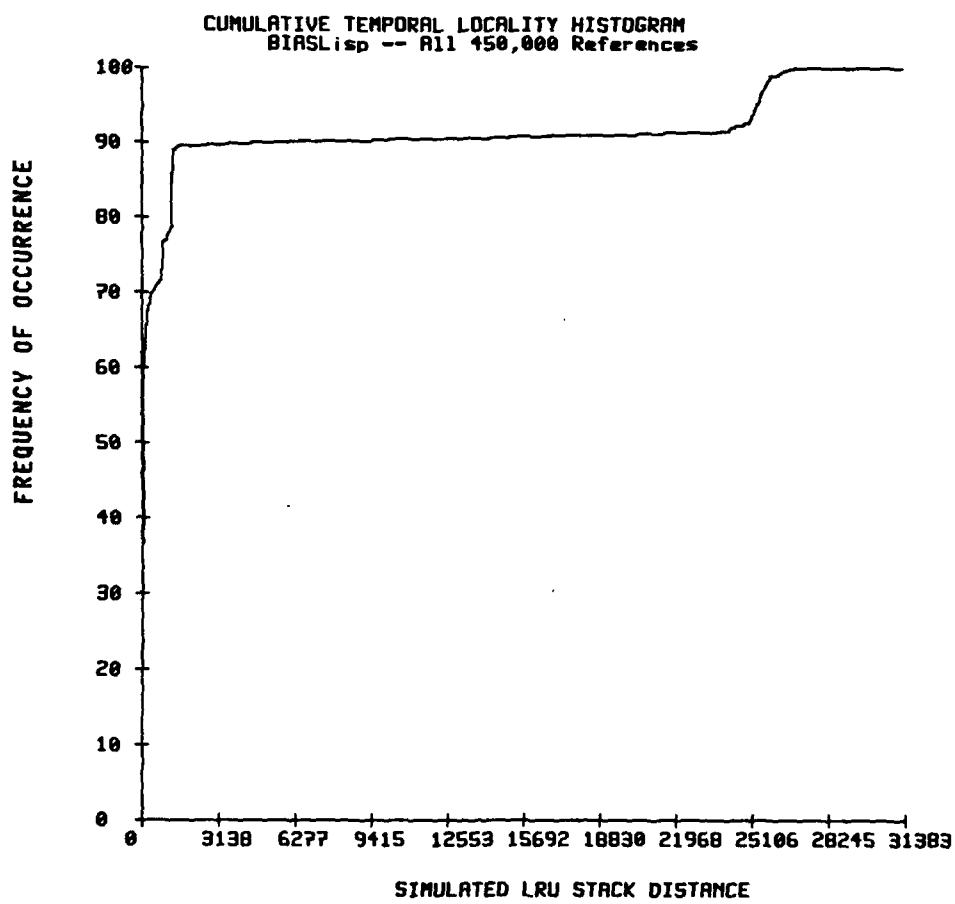


Figure 5.1: BIASLisp Cumulative Temporal Locality Histogram

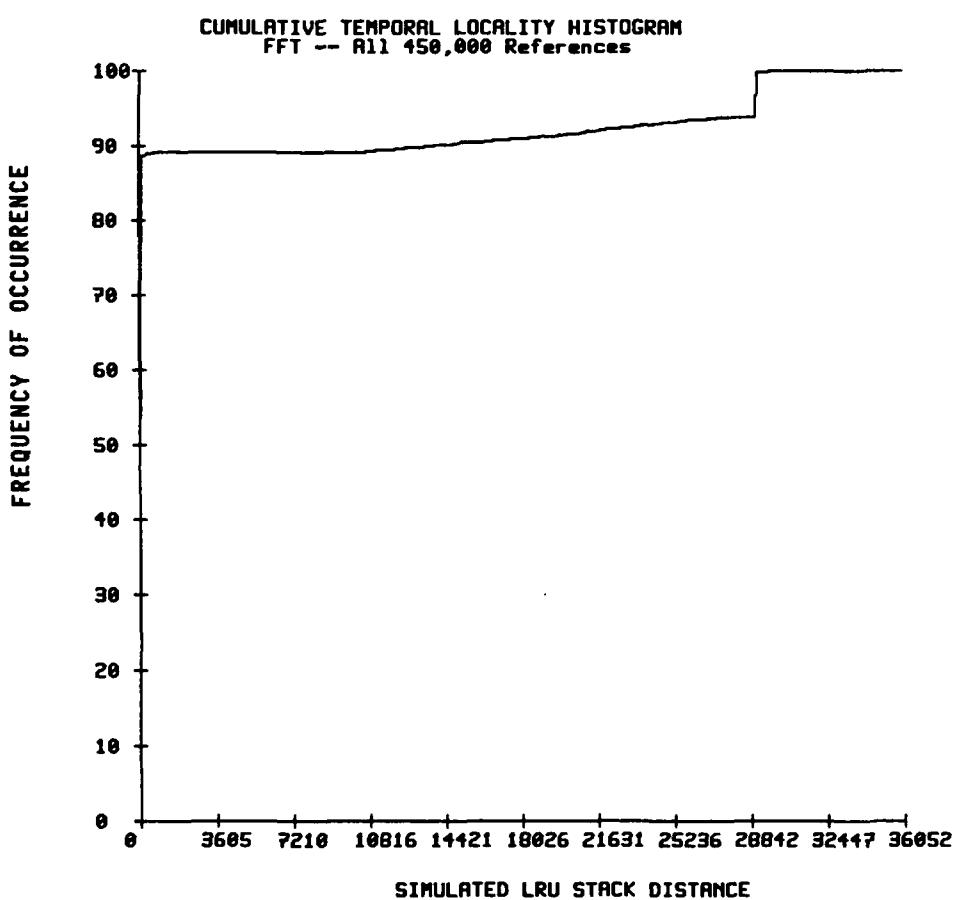


Figure 5.2: FFT Cumulative Temporal Locality Histogram

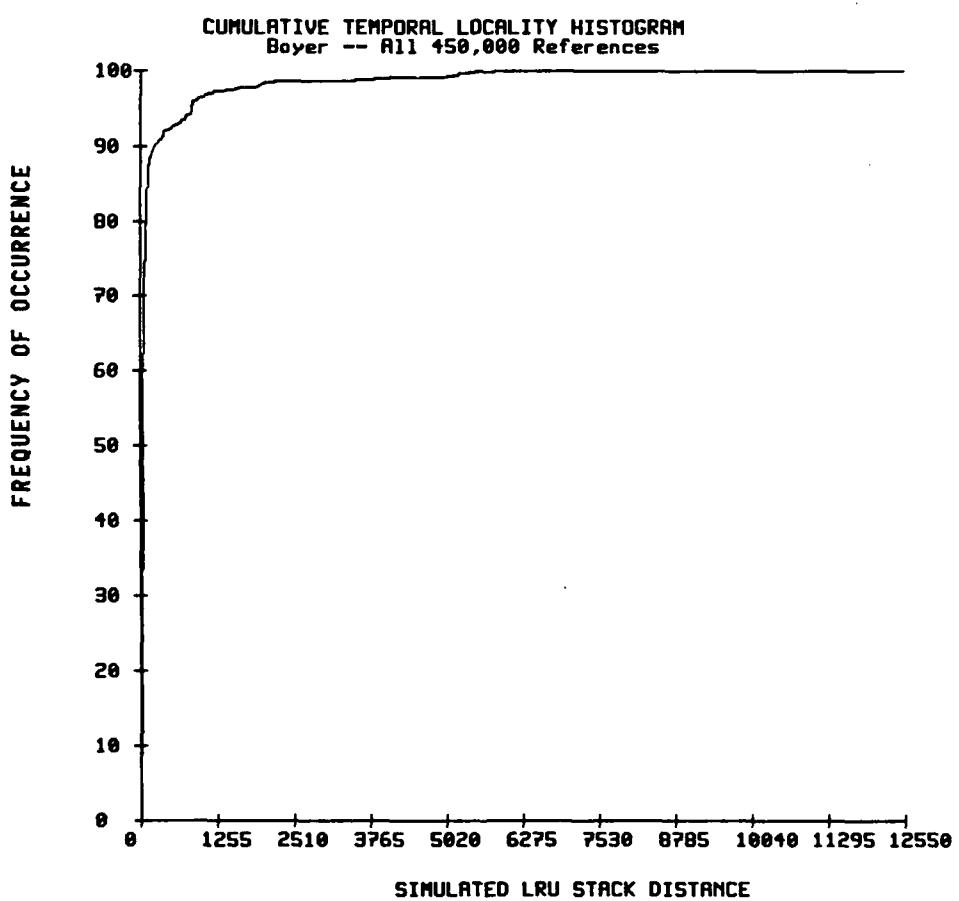


Figure 5.3: Boyer Cumulative Temporal Locality Histogram

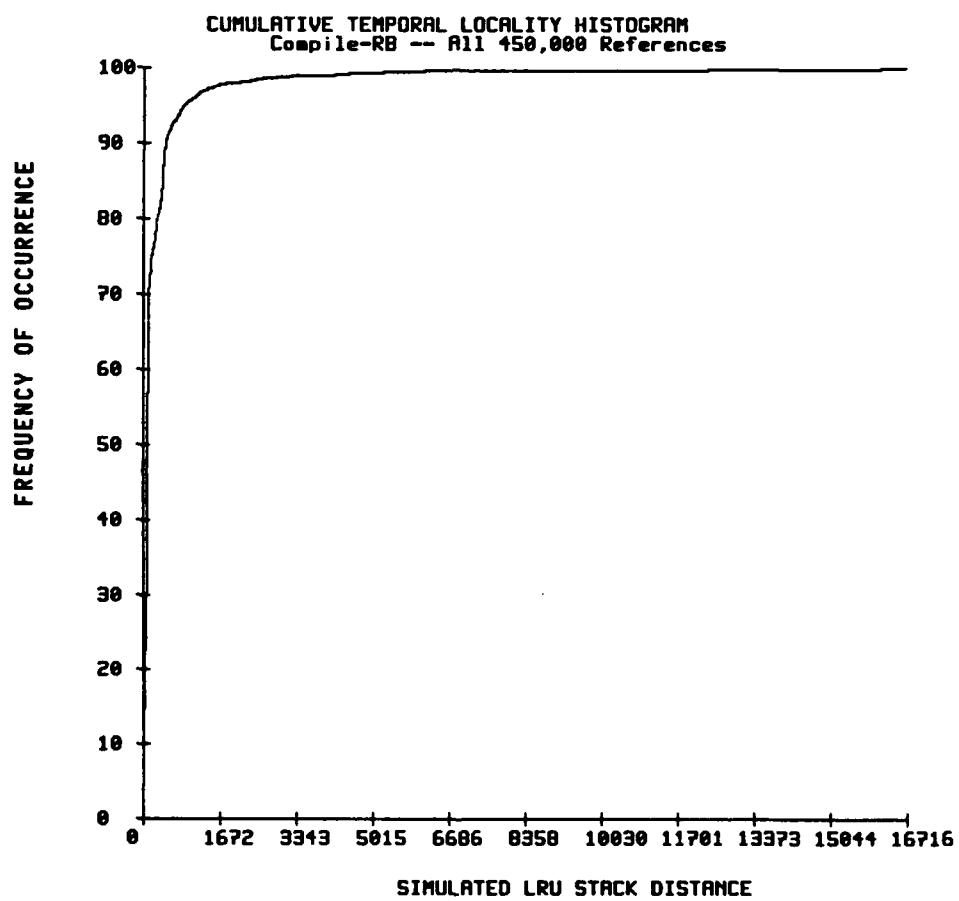


Figure 5.4: Compile-RB Cumulative Temporal Locality Histogram

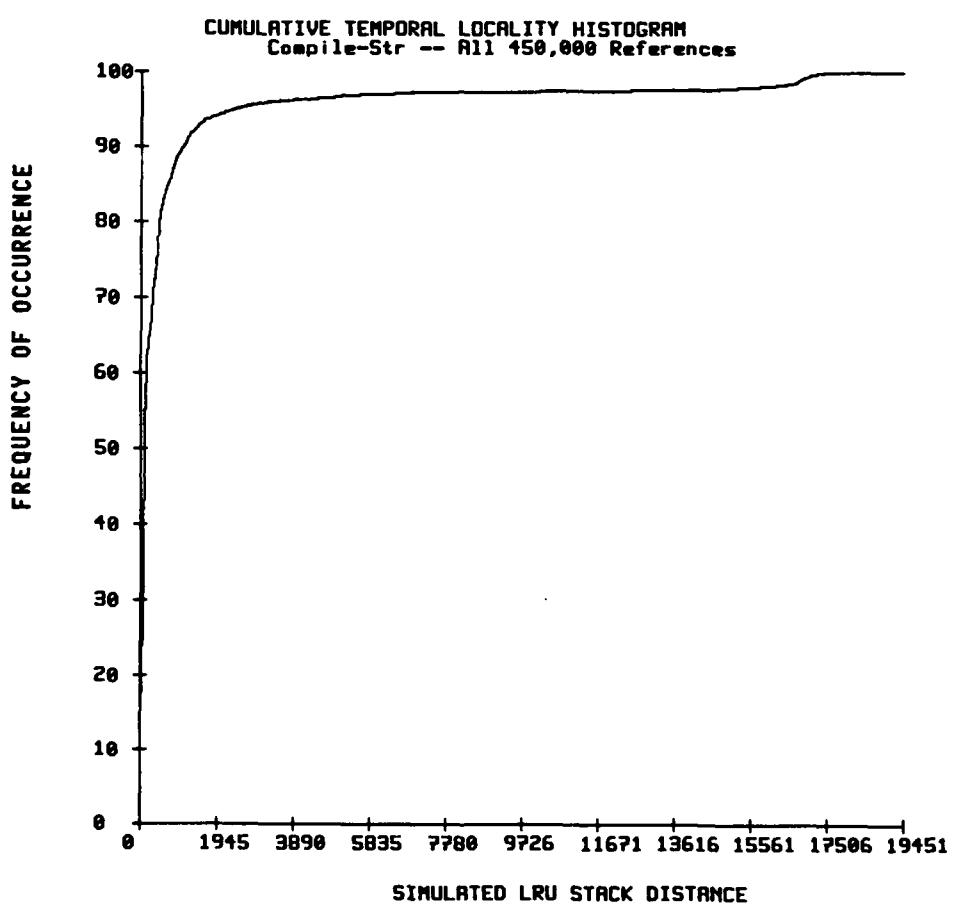


Figure 5.5: Compile-Str Cumulative Temporal Locality Histogram

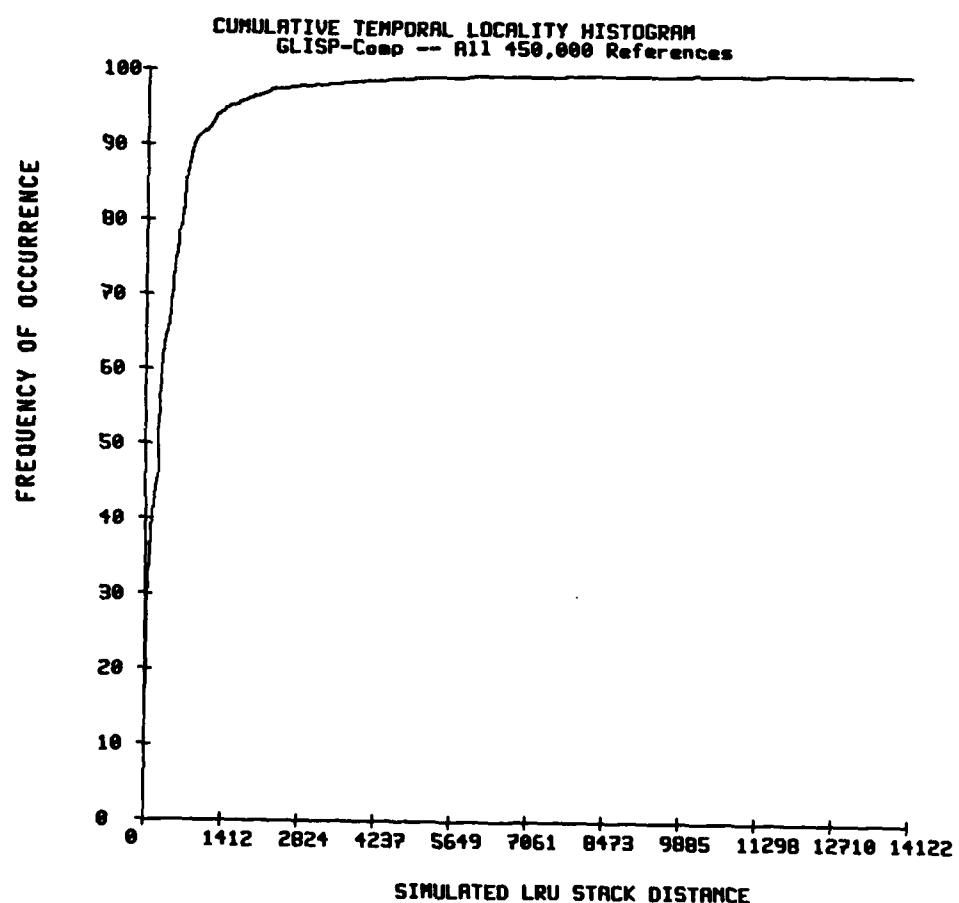


Figure 5.6: GLISP-Comp Cumulative Temporal Locality Histogram

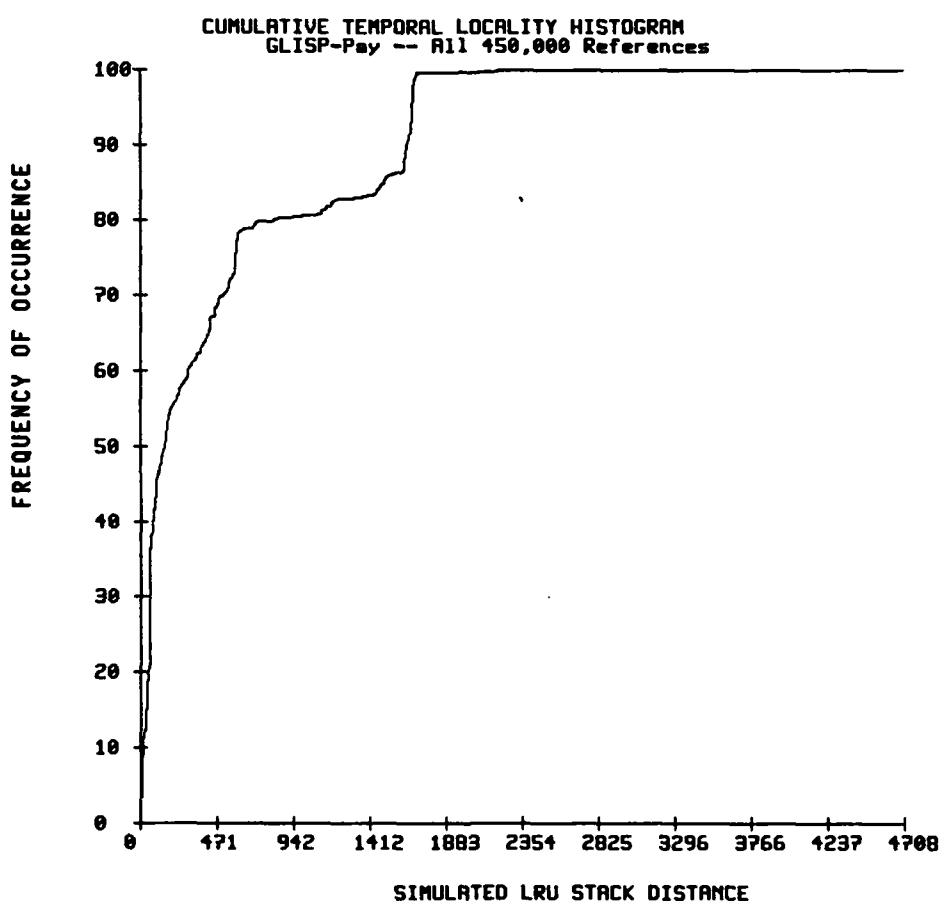


Figure 5.7: GLISP-Pay Cumulative Temporal Locality Histogram

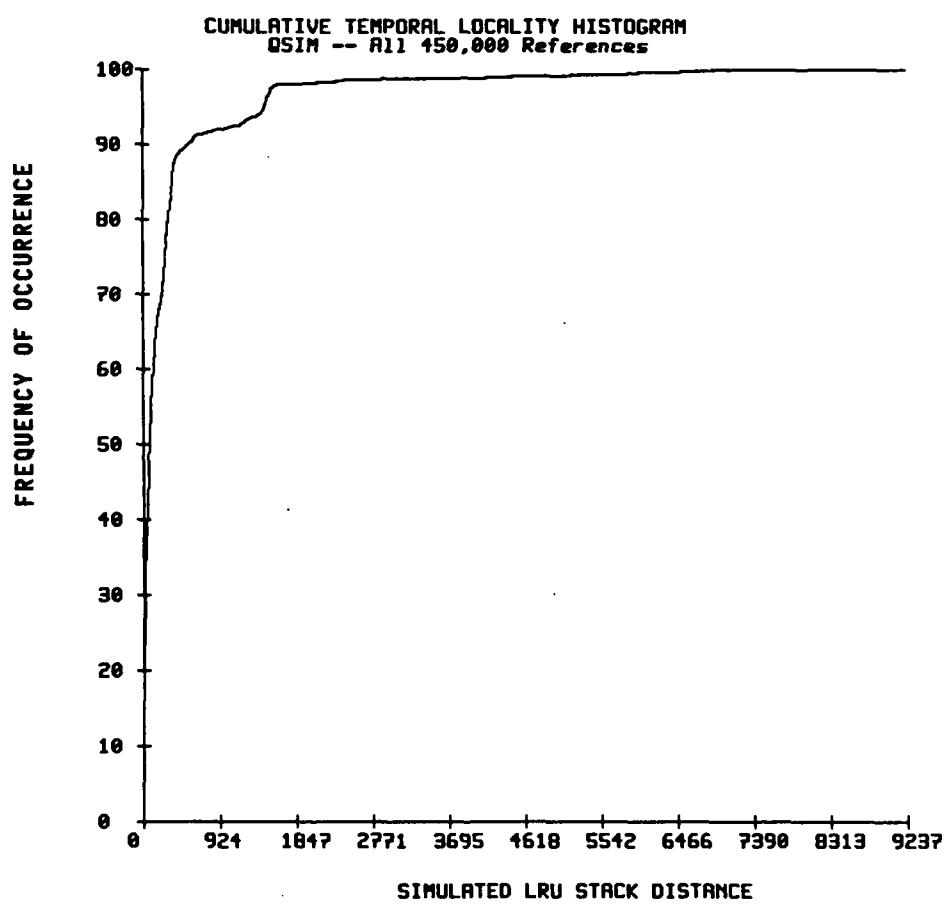


Figure 5.8: QSIM Cumulative Temporal Locality Histogram

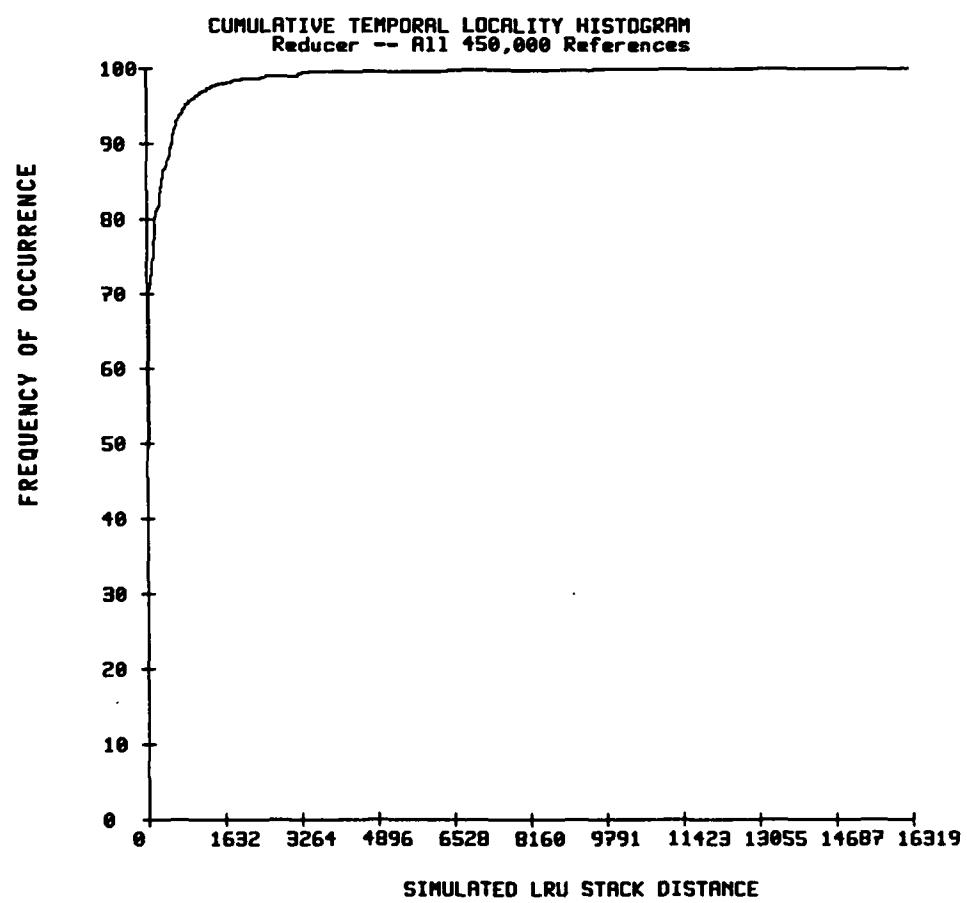


Figure 5.9: Reducer Cumulative Temporal Locality Histogram

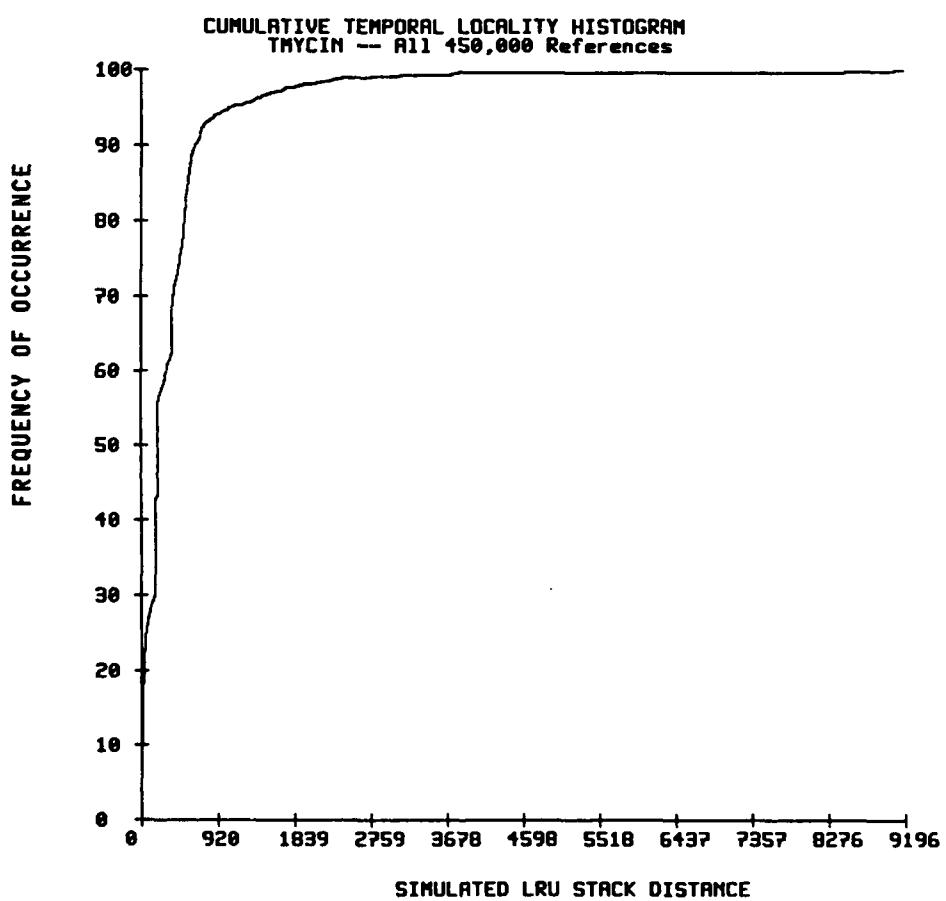


Figure 5.10: TMYCIN Cumulative Temporal Locality Histogram

Table 5.2: LRU_{90} Stack Distance Thresholds

Type Ref	Symbolic		Conventional		Ratios		t-Test Used	Conf Level
	Mean	SD	Mean	SD	Mean	F		
All	734	438	2,059	829	0.356	3.58	Eql	0.010
Inst	322	254	196	236	1.644	1.16	Eql	0.543
Data Refs	519	206	15,197	1,822	0.034	78.3	Uneql	0.055
Data Reads	512	174	24,053	477	0.021	7.46	Uneql	0.007
Data Writes	89	46	23,306	2,271	0.004	2,426	Uneql	0.044

Table 5.3: LRU_{95} Stack Distance Thresholds

Type Ref	Symbolic		Conventional		Ratios		t-Test Used	Conf Level
	Mean	SD	Mean	SD	Mean	F		
All	1,283	494	27,207	2,730	0.047	30.5	Uneql	0.046
Inst	547	367	196	236	2.791	2.41	Eql	0.244
Data Refs	788	248	26,913	3,008	0.029	147	Uneql	0.051
Data Reads	790	212	25,585	2,005	0.031	89.6	Uneql	0.036
Data Writes	184	150	23,436	2,098	0.008	194	Uneql	0.040

LRU_{95} and LRU_{99} for the conventional workloads are significantly greater than those of symbolic workloads. Table 5.2 shows that LRU_{90} is also significantly greater for the overall temporal distance strings of the conventional workloads. Although, as the ratio of the mean values of LRU_{90} for the symbolic and conventional workloads show, the difference between them is less dramatic than the differences in Tables 5.3 and 5.4 for LRU_{95} and LRU_{99} . Tables 5.2, 5.3, and 5.4 show that the likelihood that the stack distance threshold values for the overall temporal distance strings of the symbolic workloads and the threshold values for the overall temporal distance

Table 5.4: LRU_{99} Stack Distance Thresholds

Type Ref	Symbolic		Conventional		Ratios		t-Test Used	Conf Level
	Mean	SD	Mean	SD	Mean	F		
All	4,881	4,896	27,593	2,271	0.177	4.65	Eql	0.000
Inst	1,497	2,154	299	366	5.014	34.7	Eql	0.474
Data Refs	3,539	2,880	27,310	2,521	0.130	1.31	Eql	0.000
Data Reads	3,246	2,724	26,019	1,440	0.125	3.58	Eql	0.000
Data Writes	1,293	1,095	23,516	2,007	0.055	3.36	Eql	0.000

strings of the conventional workloads came from distributions with the same mean is less than 0.05 for each of the stack distance thresholds. Furthermore, the mean value of LRU_{95} for the conventional workloads is over 20 times as great.

As shown in Tables 5.2 through 5.4, the instruction reference stack distance thresholds for the conventional symbolic workloads do not differ significantly (due to the large standard deviations of the sample distributions relative to their means), whereas the data reference stack distance thresholds do. Furthermore, as shown in the above tables, when data reads and data writes are analyzed separately, both of these temporal distance strings exhibit the same locality characteristics as the overall data temporal distance strings.

The explanation for this behavior can be found in two of the high-level program characteristics of symbolic workloads. Conventional workloads tend to traverse their structures accessing the entire structure evenly. In contrast, while symbolic programs often access entire lists when they reference them for the first time, subsequent accesses to the list are concentrated on the first few elements in a list, as noted in Chapter 2. The other difference can be seen from Table 4.1. The number of distinct virtual memory addresses referenced by the symbolic workloads for data references

is only about one-third the number referenced by the conventional workloads. Thus, both the total number of distinct locations referenced and the distribution of the references over the data structures used by the programs account for this significant difference in the temporal locality stack thresholds.

5.2.3 Implications for Symbolic Memory System Design

The primary approach for exploiting the special temporal locality characteristics of symbolic workloads is to trade off cache size for other design options. Since as shown in Table 5.4, a 99 percent cache hit rate on old references can be achieved with about one fifth the cache size required for an equivalent hit rate for the conventional workloads, other parameters such as cache speed and complexity can be enhanced.

5.3 Structural Locality

5.3.1 Structural Locality Metric

To compare the structural locality of symbolic and conventional workloads, the percentages of successive references to the same position in the simulated LRU stack, P_{SSD} , was used. Thus P_{SSD} is defined as:

P_{SSD} : the probability that the subsequent reference will have the same stack distance as the previous reference given that the previous reference was an old reference

As explained in the previous chapter, these successive references having the same stack distance constitute a rereferencing of the same set of memory locations in the same order as they were last referenced. So, P_{SSD} gives an overall indication of how much of the memory referencing can be characterized by this pattern of behavior.

Table 5.5: P_{SSD} Transition Probabilities

Type Ref	Symbolic		Conventional		Ratios		t-Test Used	Conf Level
	Mean	SD	Mean	SD	Mean	F		
All	0.550	0.065	0.293	0.033	1.881	3.86	Eql	0.00078
Inst	0.923	0.044	0.992	0.010	0.930	19.9	Eql	0.06663
Data Refs	0.709	0.060	0.374	0.021	1.894	8.09	Eql	0.00007
Data Reads	0.788	0.044	0.700	0.030	1.127	2.07	Eql	0.02936
Data Writes	0.972	0.013	0.831	0.010	1.170	1.60	Eql	0.00000

5.3.2 Differences in Symbolic and Conventional Workload Structural Locality

As shown in Table 5.5, the values of P_{SSD} differ significantly between symbolic and conventional workloads. For individual symbolic workloads, P_{SSD} ranged from 0.444 to 0.626 giving the mean of 0.550 shown in Table 5.5. In other words, slightly over half of the symbolic memory references can be characterized as referencing structures. With the conventional workloads measured, the mean value of P_{SSD} is 0.293, only slightly more than half the mean P_{SSD} for symbolic workloads. Thus the percentage of memory references for conventional workloads that can be characterized as referencing structures is closer to one-fourth of the total number. Using the Student's t-Test for equal variances, one can conclude that the likelihood that the values of P_{SSD} for the symbolic workloads came from a distribution with the same mean as the distribution for conventional workloads is less than 0.0008.

The differences in structural locality are also present when the types of references are analyzed individually, except for the instruction stream. The instruction stream structural locality is extremely high for both symbolic and conventional workloads, and actually slightly higher for the conventional workloads as shown in

Table 5.5. However, for the data references, the mean structural locality of the symbolic workloads is about twice that of the conventional workloads. There are also significant differences in the mean P_{SSD} values for symbolic and conventional workloads when the data reads and data writes are analyzed individually as is also shown in Table 5.5. One can also see that the structural locality is greater when each type of reference, such as data reads, is analyzed separately than for composite temporal distance strings that are made up of all data references. The data writes have the greatest mean structural locality for the symbolic workloads with a P_{SSD} of 0.972.

At first, these results might seem counterintuitive, since, as previously mentioned, the conventional workloads have a more uniform and more regular accessing pattern for their data structures. However, while this accessing pattern is more uniform and regular, the actual order of access changes from one traversal of the main structure to the next. Because of this, a whole new working set of substructures is generated for each traversal of the main structure. With the symbolic workloads, on the other hand, the same basic substructures are usually traversed in the same order each time.

5.3.3 Implications for Symbolic Memory System Design

The structural locality of the symbolic workloads provides possibly the greatest opportunity for exploiting the unique referencing behavior of this type of workload. As mentioned in the background chapter, there have been other attempts to exploit the structural locality through novel list representations and caching schemes. The method proposed in this dissertation involves mapping the structural locality of the low-level memory referencing behavior to a spatial locality in a fully-associative main cache and then prefetching from the main cache to a smaller on-chip cache. Chapter 7 details this design as well as providing an analytic model

to evaluate its performance.

5.4 State Transition Probabilities

5.4.1 Metrics Used for the Comparison

There are two other important metrics for comparing conventional and symbolic workloads: the frequency of program transitions from referencing previously-referenced locations—the Old-Ref state—to referencing previously-unreferenced locations—the New-Ref state—and vice versa. These metrics are thus defined as follows:

P_{ON} : the probability that the subsequent reference will be a new reference given that the previous reference was an old reference

P_{NO} : the probability that the subsequent reference will be an old reference given that the previous reference was a new reference

These are the Old-New and New-Old transition probabilities described in the previous chapter. Together, these metrics reveal two important characteristics of the workloads: first, how often the workload will transition to the New-Ref state, and once there, how long it will stay in the New-Ref state.

5.4.2 Differences Between Symbolic and Conventional Workloads

The mean Old-New transition probability, P_{ON} , for the conventional workloads is over four times greater than the mean P_{ON} of the symbolic workloads. And, using the Student's t-Test for equal variances, one can conclude that the likelihood that the transition probabilities for the symbolic workloads and those for the conventional workloads came from distributions with the same mean is only 0.00002.

As Table 5.6 shows, this difference in the Old-New transition probabilities was analyzed for each type of memory reference. For instruction fetches, the Old-

Table 5.6: *P_{ON}* Transition Probabilities

Type Ref	Symbolic		Conventional		Ratios		t-Test Used	Conf Level
	Mean	SD	Mean	SD	Mean	F		
All	0.013	0.0063	0.060	0.0078	0.223	1.50	Eql	0.000
Inst	0.001	0.0008	0.000	0.0000	0.000	0.00	Uneql	0.021
Data Refs	0.011	0.0062	0.059	0.0078	0.190	1.59	Eql	0.000
Data Reads	0.017	0.0108	0.054	0.0057	0.306	3.66	Eql	0.002
Data Writes	0.007	0.0064	0.051	0.0085	0.132	1.72	Eql	0.000

New transition probabilities are extremely low in both symbolic and conventional workloads. However, for the entire data reference string, as well as for the individual data read and write reference strings, there are significant differences. The greatest difference is in the data write reference string where the mean Old-New transition probability for the conventional workloads is over seven times that of the symbolic workloads.

This difference between conventional and symbolic workloads is due to two factors: the smaller mean number of new data read references for the symbolic workloads and the different ways in which these types of workloads access their data structures. As shown in Table 4.1, the mean number of new data read references for the symbolic workloads is only one-third the mean number of new data read references for the conventional workloads. In addition, the conventional workloads usually access each data structure element individually, do some processing on it, and then write back the result of that processing. On the other hand, while symbolic workloads read their list elements individually, they write the results of a list operation out to memory through one transition to the New-Ref state and then by making subsequent new references until the entire result list has been written. Thus,

Table 5.7: P_{NO} Transition Probabilities

Type Ref	Symbolic		Conventional		Ratios		t-Test Used	Conf Level
	Mean	SD	Mean	SD	Mean	F		
All	0.371	0.124	0.696	0.015	0.534	69.2	Eql	0.008
Inst	0.148	0.048	0.069	0.009	2.161	27.0	Eql	0.055
Data Refs	0.399	0.140	0.694	0.028	0.574	24.6	Eql	0.022
Data Reads	0.631	0.171	0.670	0.024	0.941	50.4	Eql	0.763
Data Writes	0.359	0.174	0.684	0.021	0.525	72.2	Eql	0.036

rather than transitioning from the Old-Ref state to the New-Ref state each time a list element has to be written, one transition is normally made, and the entire list is written.

A significant difference between symbolic and conventional workloads in the New-Old transition probabilities, as shown in Table 5.7, supports the above explanation. The mean symbolic workload New-Old transition probability is only about half that of the conventional workloads. Analyzing just the data reads and data writes shows that there is no significant difference in the data read mean P_{NO} probabilities for the symbolic and conventional workloads, but the mean data write P_{NO} probability for the symbolic workloads reflects the difference present in the overall memory reference stream. Thus, the P_{NO} transition probabilities substantiate the assertion that this difference in the P_{ON} transition probabilities is due, at least in part, to the consecutive new references made to write back to memory the results of list operations.

5.4.3 Implications for Symbolic Memory Subsystem Design

Any memory design exploiting these state transition probability differences must focus on the smaller total number of new data read references and the tendency of symbolic workloads to write the results of lists operations out to memory through consecutive new memory references. The smaller number of new data read references will tend to decrease the impact of prefetching, since the overall cache miss rate due to new references will be less for symbolic workloads. However, the use of a buffered write-back scheme rather than a write-through cache scheme should decrease the effective memory access time.

5.5 Results of Correlogram and Power Spectrum Analysis

While the correlograms and power spectra were analyzed, no significant differences in the periodicity or order characteristics of symbolic and conventional workloads were found. Instead, these plots seemed to depend much more on the type of reference (e.g., instruction fetch or data ref) and type of locality (e.g., temporal or spatial) being characterized than whether the workload was a symbolic or a conventional one. Temporal distance strings had decreasing autocorrelation with increasing lag in accordance with Denning's definition of locality in Chapter 2. However, spatial distance strings did not follow this pattern. Instead, the autocovariance varied over short periods of time but did not decrease in average value even for large lags. This was reflected in all the autocovariance functions and power spectra of the spatial and temporal distance strings for both symbolic and conventional workloads and is typified by the plots shown in Figures 5.11 through 5.14.

The FFT workload provided some interesting plots due to the workload's very ordered data structure accessing pattern. The regularity of this accessing pattern is clearly shown by the correlogram and power spectrum of the FFT data write

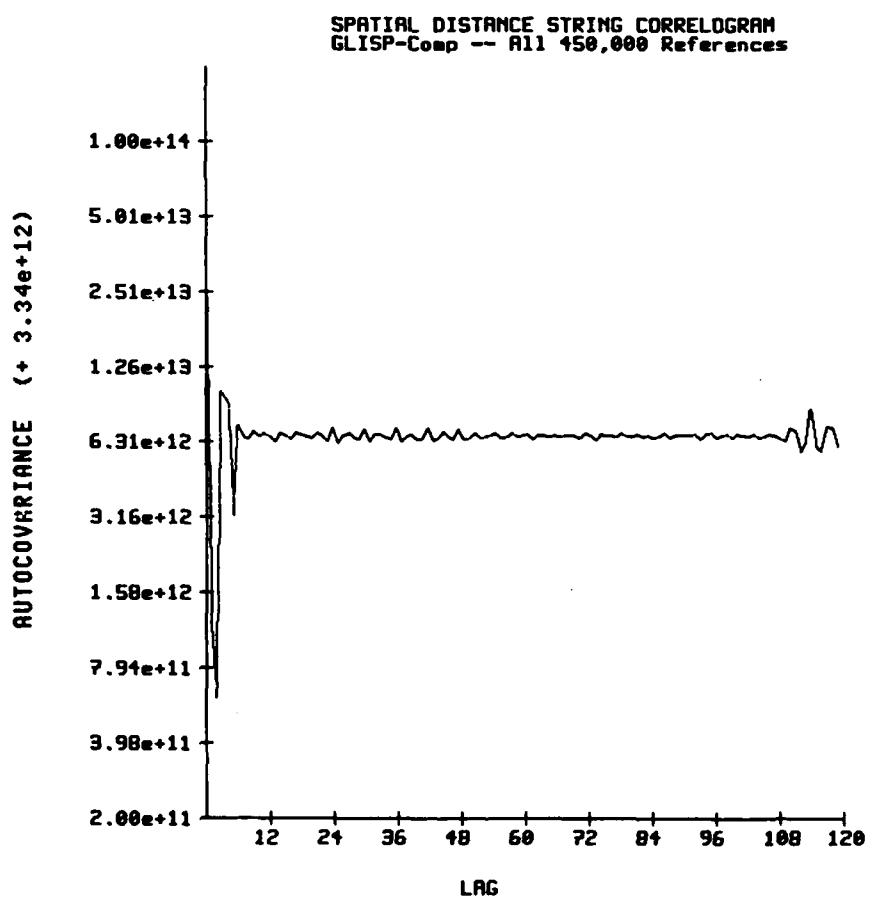


Figure 5.11: Example Correlogram of a Spatial Distance String

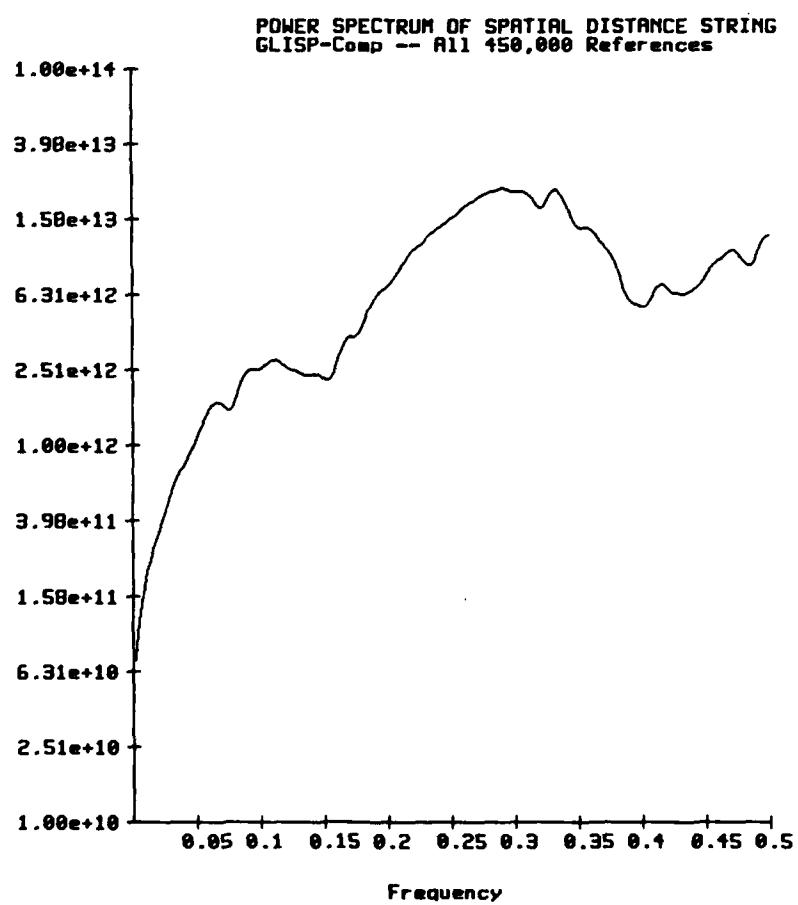


Figure 5.12: Example Power Spectrum of a Spatial Distance String

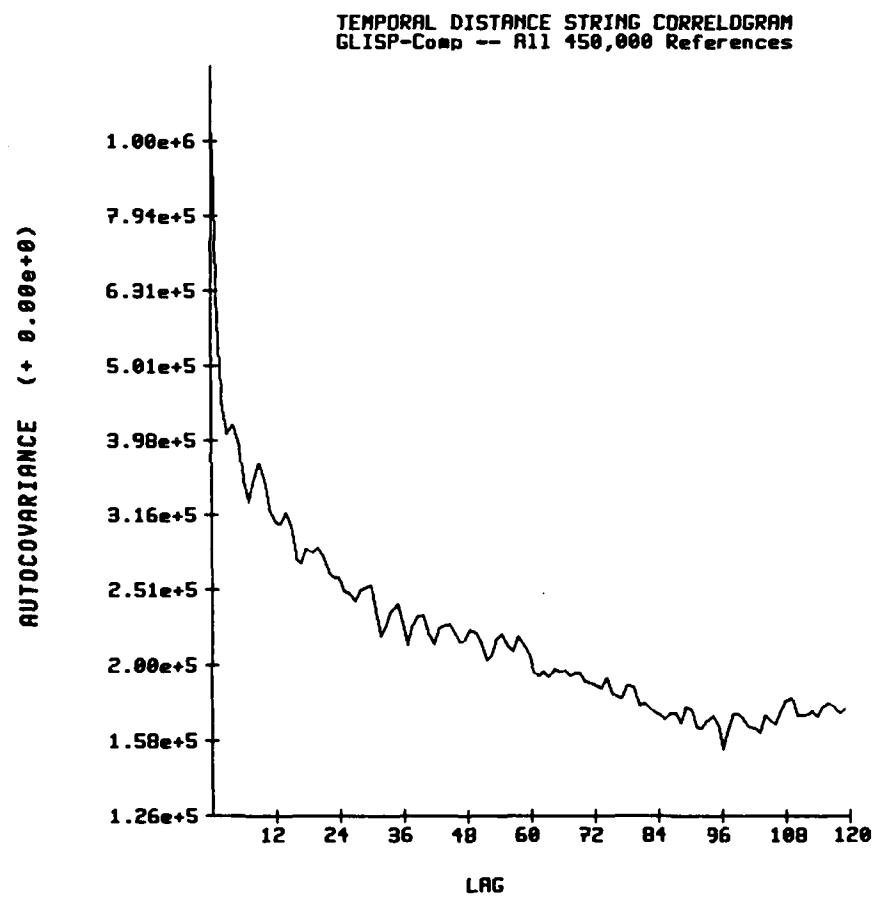


Figure 5.13: Example Correlogram of a Temporal Distance String

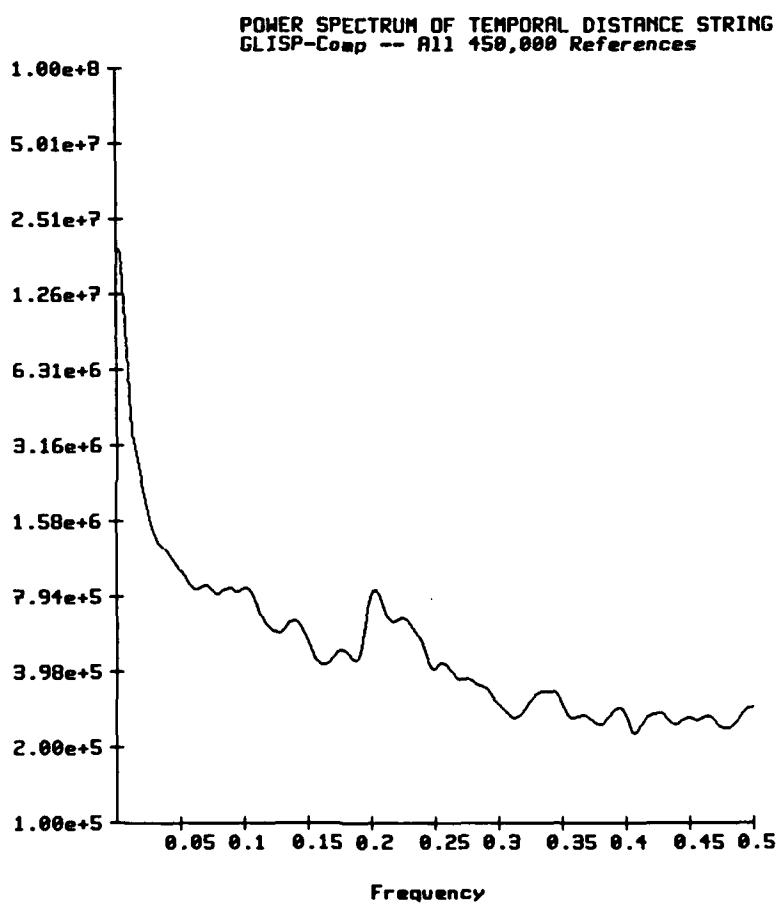


Figure 5.14: Example Power Spectrum of a Temporal Distance String

New-New reference string shown in Figures 5.15 and 5.16.

5.6 Summary

This research has identified several significant differences in the locality characteristics of symbolic and conventional workloads. The spatial locality of symbolic workloads, as quantified by the spatial window probability, P_{SW} , is greater than that of conventional workloads. The stack distance thresholds, LRU_{95} and LRU_{99} , are much smaller for symbolic workloads. The symbolic workloads have greater structural locality, as quantified by P_{SSD} , and finally the Old-New transition probability is much lower for symbolic workloads. Principal reasons for these differences include the greater number and smaller size of functions in symbolic workloads, the inherent difference in the way data structures are accessed for these two types of workloads, and the smaller number of distinct memory locations referenced by the symbolic workloads.

However, it is not the intent of this chapter to imply that these are the only differences between symbolic and conventional workloads. Computation of other locality characteristics and even further analysis of the characteristics computed in this research may provide additional insight into these two types of workloads.

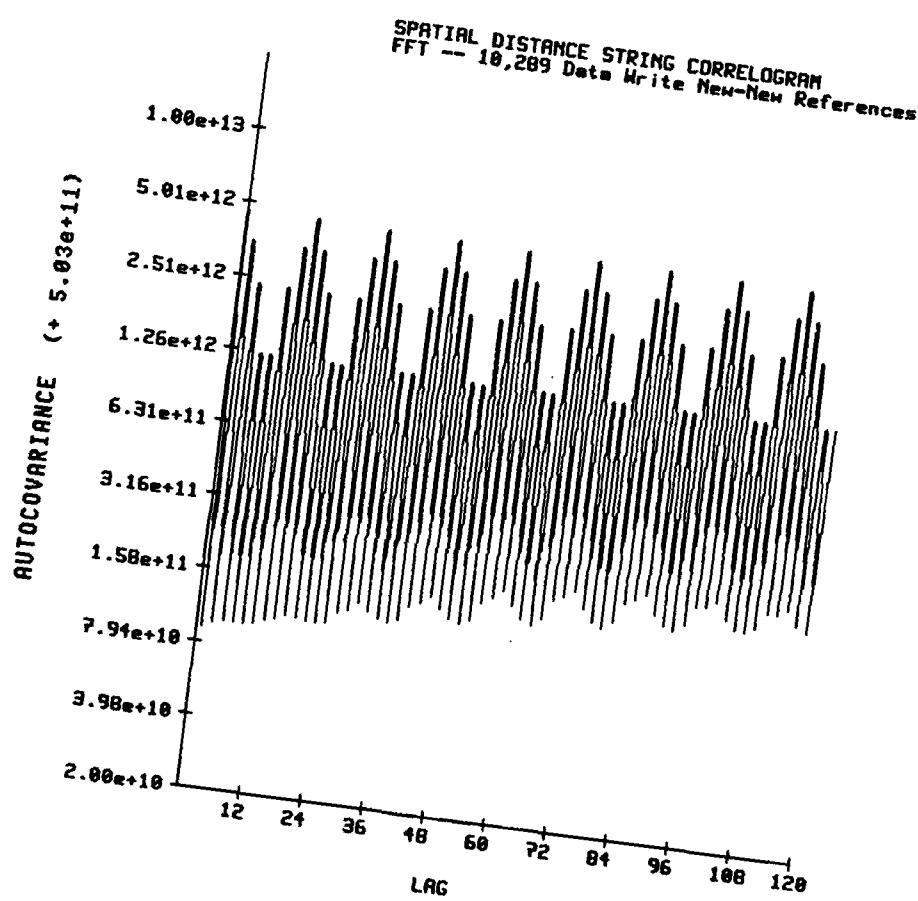


Figure 5.15: Correlogram of the FFT Data Write New-New Spatial Distance String

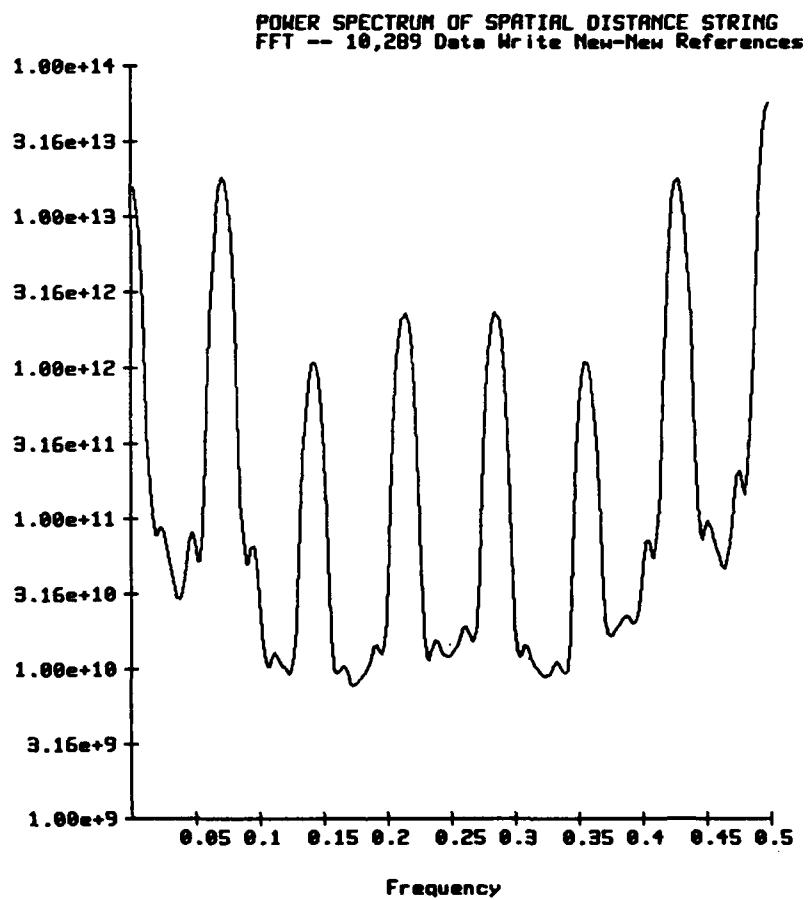


Figure 5.16: Power Spectrum of the FFT Data Write New-New Spatial Distance String

Chapter 6

Validation of Results

This chapter describes the various methods used to validate the results of the previous chapter. It first describes the validation of the software routines used to generate the results and then details the techniques used to validate the results themselves. These techniques include the generation of test plots with highly structured data, comparison with previously published results, and cross-checking the results for consistency and reasonableness. To assess the influence of the Lisp Machine architecture on the results, a two-fold approach was used. First, each of the major factors that could influence the results was analyzed and its probable impact assessed. Then the traces of programs executed on an IBM System/360 Model 91 at the Stanford Linear Accelerator and used in several cache performance studies such as [Smit82] were used to generate locality characteristics for comparison with those in the previous chapter. The similarities of the locality characteristics for these vastly different architectures helped to validate the architectural independence of some of the results, while the differences in the locality characteristics helped to assess the influence of the various architectural differences.

6.1 Validation of the Software Routines

Each routine was tested extensively, both before integration into the trace analysis software, and after its integration. Each routine was written in Lisp code and the only global variables accessed were the print base and the window name for

the plots. To insure that the sorting routines were generating the proper memory reference strings, a completely decoded portion of one of the virtual memory address traces was used to verify that each of sorted traces contained the correct subset of the memory references.

To test the autocovariance and Fourier transform routines, very regular strings whose output could be easily predicted, such as the one below,

1 1 9 1 1 9 1 1 9 ...

were used as input to these routines. These routines were also used to duplicate correlogram and power spectrum plots Spirn had published in his book [Spir77].

Once the results were generated, they were cross-checked for consistency. These checks are listed below:

1. Does the sum of the numbers of references in each of the traces sorted by memory reference type (e.g., instruction fetch) equal 450,000?
2. Is the number of Old-New references within one of the number of New-Old references?
3. Are the percentages of each type of memory reference reasonable when compared with other published results such as those cited in Chapter 2?
4. Is the maximum stack distance of each temporal distance string less than the string length?

6.2 Evaluation of Architectural Independence of Results

This evaluation consisted of two complementary approaches. The first approach consisted of evaluating those features of the Explorer II architecture which

could affect the results. The second approach was to compare the locality characteristics of the Explorer II traces with those computed from the traces of workloads on an IBM System/360 Model 91.

6.2.1 Distinctive Architectural Features of the Explorer II

Altogether, five principal architectural features of the Explorer II were considered. One of these features, was the Explorer II's use of *cdr*-coding to exploit the spatial locality of list structures. The principal effect of a frequent use of *cdr*-coding would be a spatial distance of one word rather than two words for consecutive list elements. As mentioned in the background chapter, Clark showed in his studies that list elements are placed close to each other without *cdr*-coding or a special *cons* algorithm. Instead, because the elements of a list are usually created temporally close together, the memory locations allocated from the heap memory are normally spatially contiguous or at least very close. Although *cdr*-coding is used only for *list*, *append*, and *reverse* commands on the Explorer II and not for regular *cons* operations, it is used to write out lists which have been built in the PDL and so access to *cdr*-coded structures is fairly common for the symbolic workloads. Even so, since none of the results mentioned in the previous chapter would be affected by whether the distance between successive elements of a list was one word or two, *cdr*-coding did not influence these results.

The PDL, which was just mentioned, is a 1000-word top of stack on-chip buffer, and its influence on the results must also be considered. The PDL is used in place of a register set, with most local scalar variables or pointers to these local variables being stored in this buffer. Since this buffer is large, it also eliminates the virtual memory references occurring from register overflows in conventional architectures with smaller register sets. However, all data structures, compiled code,

and global variables are stored in the heap memory. Thus, the results of the previous chapter characterize the program referencing behavior of the workloads' data structures and not the scalar referencing within a function nor parameter passing between functions. As mentioned in the background chapter, there has been one attempt, trace flattening, to incorporate the register set into the overall memory hierarchy for temporal locality analysis. This technique was not used in my research although it may provide a way in future work to incorporate the register set in a further characterization of the temporal and structural locality of symbolic workloads. However, there is no way to incorporate the register set into a characterization of the spatial locality of the workloads without first defining a distance measure for the register set and assigning it a space in the virtual memory. And even incorporating the register set does not eliminate all influence of the architecture. For instance, intermediate results which might be stored in register memory for one architecture might be stored within the arithmetic logic unit for another processor architecture (e.g., software versus hardware multiplies).

This suggests another of the features of the Explorer II, its heavily-micro-coded complex instruction set (CISC) architecture. Because of this architecture, one instruction fetch may cause a number of data reads or writes as in the case of writing a list out to memory or appending an element to a list.

Because of the large PDL and CISC architecture, the virtual memory references characterized by this research comprise a common subset of the virtual memory references across all simpler architectures. As such, symbolic workload word-level virtual memory referencing behavior characterized in the previous chapter is not due to architectural features such as a limited instruction set or a small register set but rather to access requirements of the workload that are common to all architectures. Thus this research can be contrasted with Mitchell's study of the influence of proces-

sor architecture on cache performance, where he focused on those references captured in the PDL [Mitc86].

However, the CISC architecture and large PDL do bias the characterization of the symbolic workloads in one respect. The buffering of intermediate list results and the writing of the list to memory through successive data write accesses lowers the New-Old transition probability, P_{NO} , for data writes, as was noted in Chapter 5. Thus, the implication that a buffered write-back algorithm is preferred over a write-through algorithm cannot be generalized for symbolic workloads on all architectures.

The Lisp compiler on the Explorer II further insures that only those virtual memory references required by all architectures are characterized. The compiler does an excellent job of using the PDL whenever possible and of minimizing virtual memory references.

The last feature of the Explorer II which had an influence on the results in the previous chapter is the size of the virtual memory address space and the memory allocation strategy used for the various kinds of virtual memory references: system variables, system Lisp routines, and user defined routines and data structures. As described earlier, each of the above is allocated in a separate portion of the virtual memory address space. And because the virtual memory address space is very large, the distance between these subspaces is substantial. However, since a large virtual memory address space is more the rule than the exception among newer architectures, as is the use of subspaces and the separation between them, this Explorer II architectural feature helps identify the virtual memory referencing behavior resulting from this trend toward larger virtual memory address spaces.

6.2.2 Comparison Between Explorer II and IBM System/360 Model 91 Traces

Although the IBM System/360 Model 91 was chosen because of the availability of the trace data and of results derived from the use of these traces, it was also a good choice for another reason. Architecturally, it is vastly different from the Explorer II and thus provides a good basis for evaluating the architectural independence of the results computed from the Explorer II traces. Specifically, the IBM System/360 Model 91 has a much smaller register set, a memory address space which is one-eighth the size, and no *cdr*-coding. The instruction set architecture is fairly complex, although simpler than that of the Explorer II.

Despite these differences, some of the Explorer II results could also be derived from the IBM System/360 Model 91 traces. The primary factor preventing a confirmation of the differences, was the lack of a trace of a true symbolic workload for the IBM System/360 Model 91. Although two of the workloads, the WATFIV FORTRAN compiler and the WATEX sorting and bin packing routine were data-driven, none of the workloads used lists.

The spatial locality characteristics observed in both conventional and symbolic workloads on the Explorer II were also present in the IBM System/360 Model 91 traces. As can be seen by comparing Figures 6.1 through 6.5 with Figure 4.4, the plateaus are present in the spatial locality cumulative histograms, although they are shorter due to the smaller memory address space and thus shorter distances between subspaces. The width of the spatial locality window was also smaller, although for comparison with the Explorer traces, P_{SW} was not redefined. As with the Explorer traces, there was a significant number of accesses, about 40 percent, within the spatial locality window. Table A-1 in Appendix A shows the very similar spatial locality results that were obtained when the same type of workload, a fast Fourier

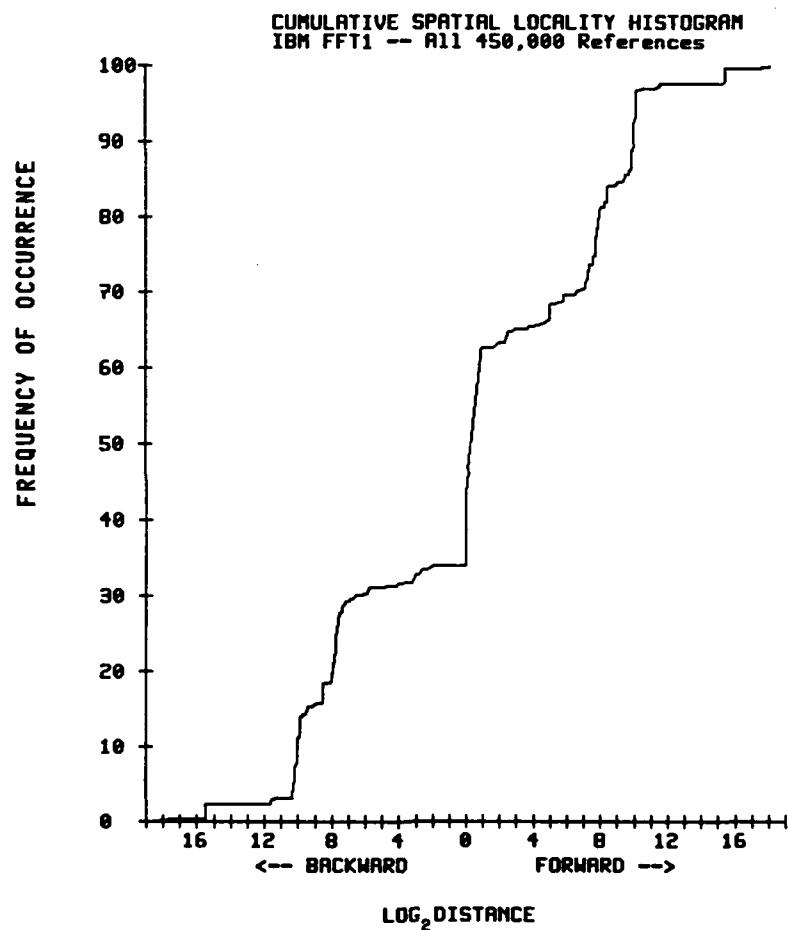


Figure 6.1: IBM FFT1 Cumulative Spatial Locality Histogram

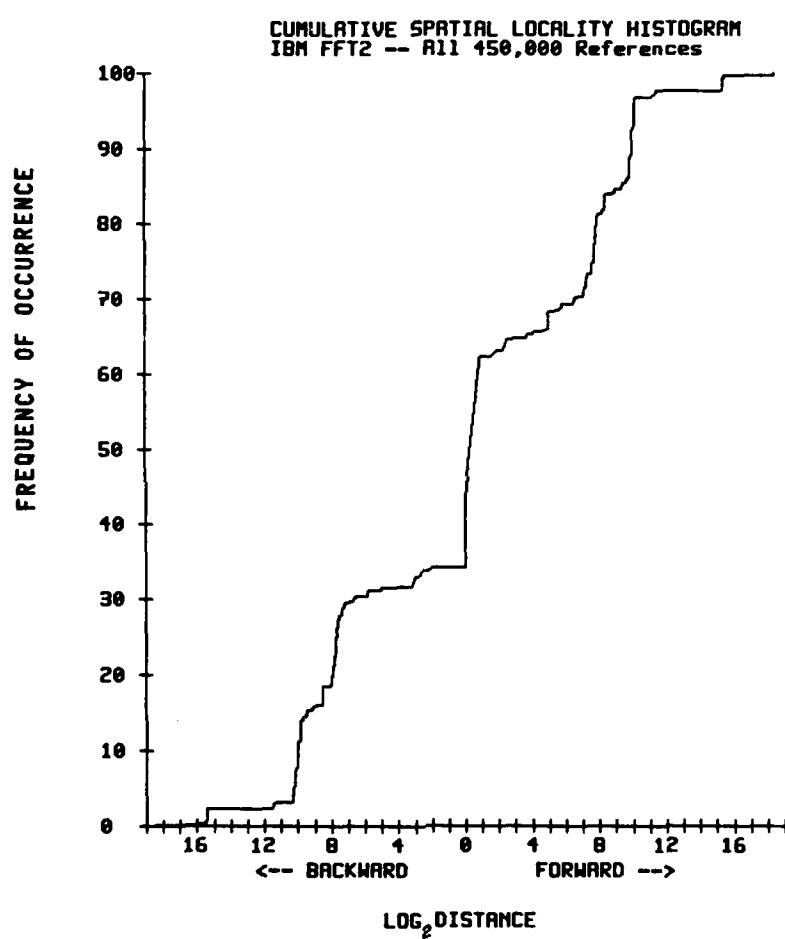


Figure 6.2: IBM FFT2 Cumulative Spatial Locality Histogram

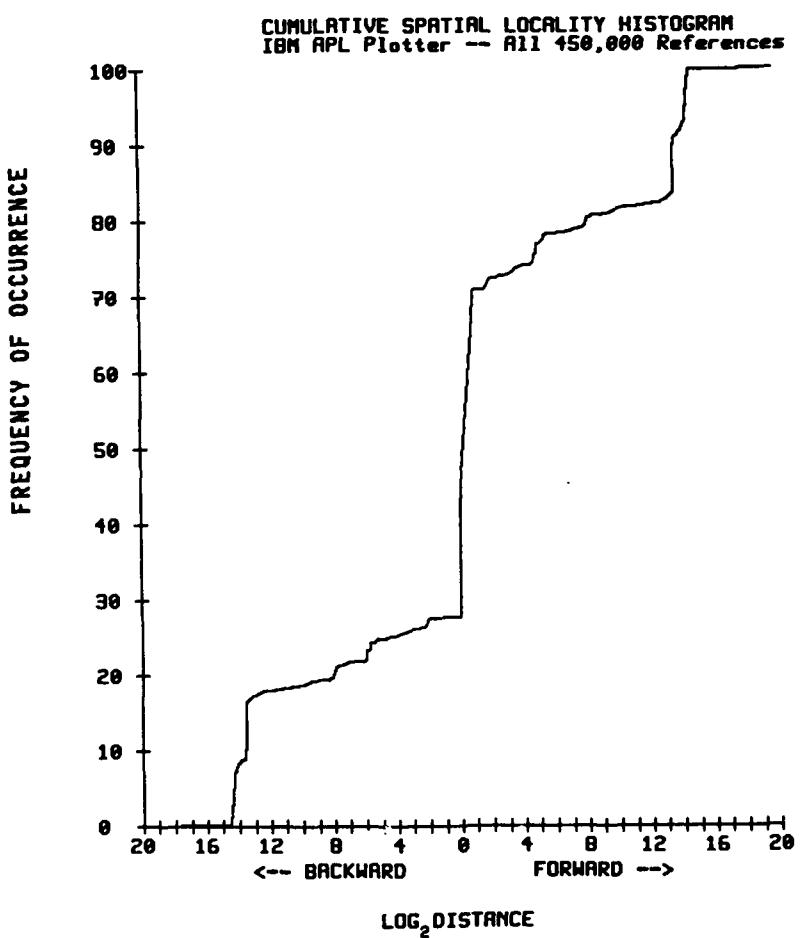


Figure 6.3: IBM APL-Plotter Cumulative Spatial Locality Histogram

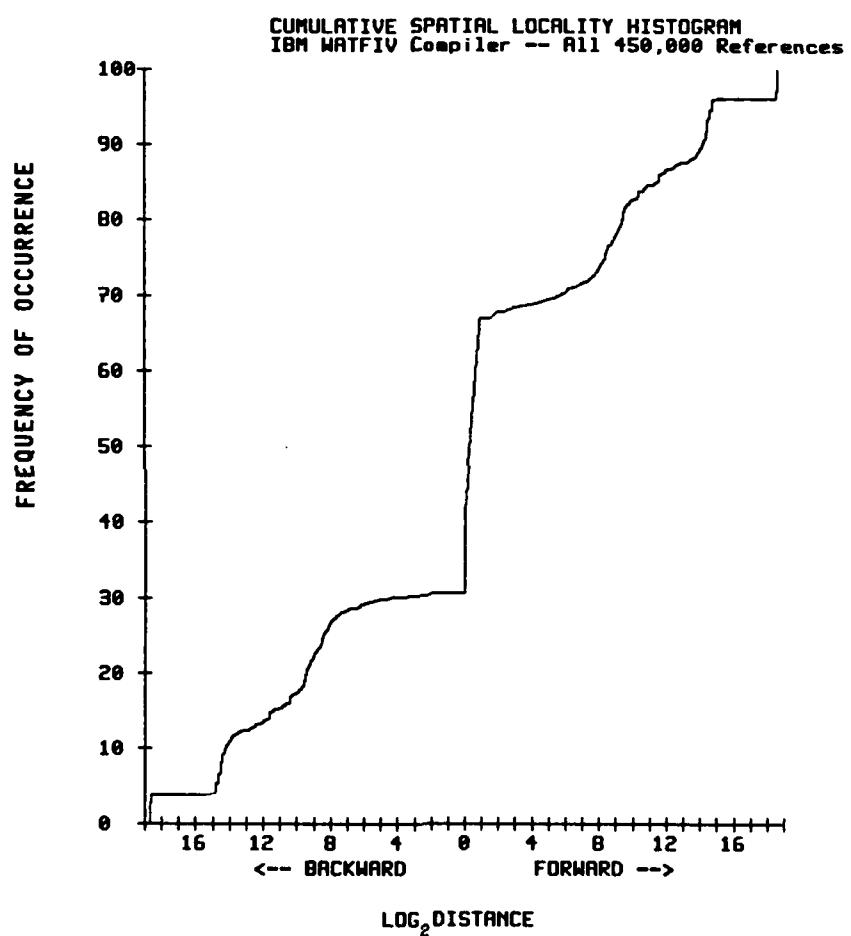


Figure 6.4: IBM WATFIV Compiler Cumulative Spatial Locality Histogram

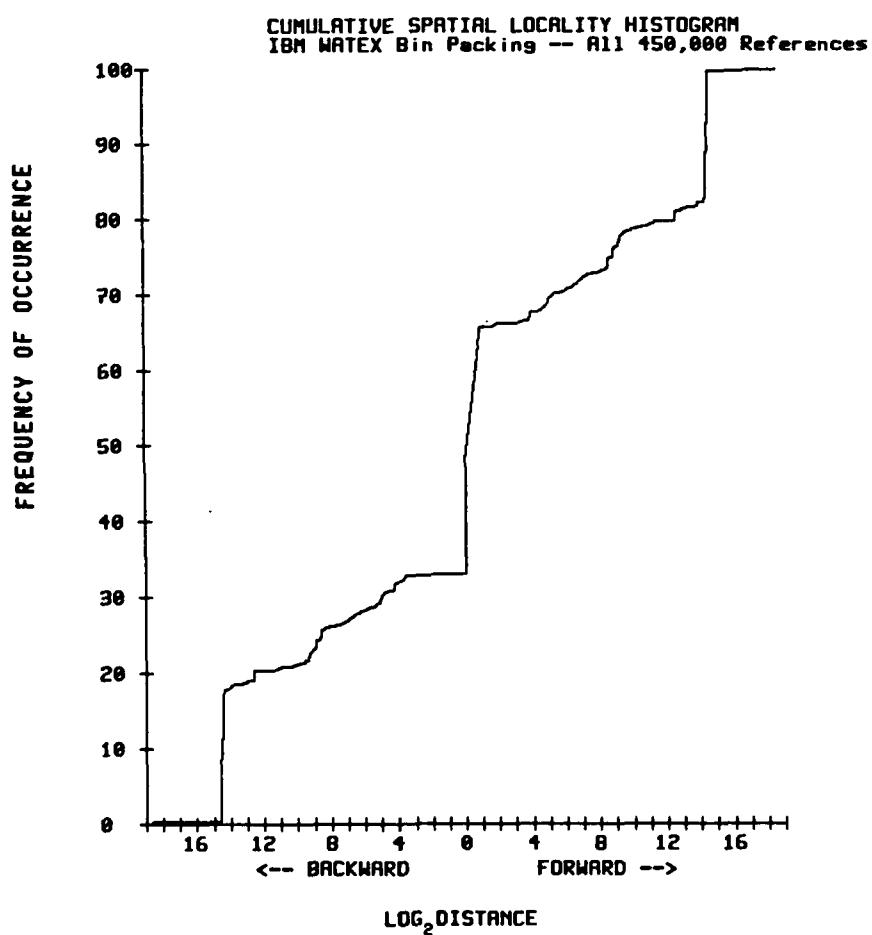


Figure 6.5: IBM WATEX Bin Packing Cumulative Spatial Locality Histogram

transform, was traced on these two vastly different architectures.

Other locality results preserved across both symbolic and conventional workloads as well as across both machine architectures were the very high spatial locality of instruction references and the very high spatial locality of instruction fetches during New-New transitions, i.e., new instruction fetches. As shown in Table A-2 of Appendix A, the standard deviations for all fifteen workloads were well under ten percent of the mean values. The high structural locality of data reads and writes was also a common characteristic for both symbolic and conventional workloads on both machine architectures as shown in Tables A-14 and A-15 of Appendix A.

Thus, this comparison helped confirm the validity of the results and show where the Explorer II architecture had influenced the results, and also show where the results were independent of the machine architecture. While it could not confirm the differences between symbolic and conventional workloads, due to the lack of true symbolic workloads on the IBM System/360 Model 91, it did indicate that a key high-level feature of symbolic workloads essential to distinguishing them from conventional workloads is their use of lists. In fact, when the results of the GLISP-Pay workload, an algorithmic rather than data-driven routine, using lists are considered, the algorithmic nature of the workload appears to have had much less influence in determining the memory referencing locality characteristics of the workload than its use of lists.

6.3 Summary

The results obtained in this research were subjected to extensive cross-checking and validation, and they were critically evaluated for biases introduced by the Explorer II architecture. Based on this analysis, it can be concluded that the symbolic workload locality characteristics and their differences from conventional

workload locality characteristics are not due to the Explorer architecture, but rather to inherent differences in the ways that these two types of workloads access memory.

Chapter 7

Application to Memory Subsystem Design

This chapter gives an example of applying the measured symbolic workload locality characteristics to a memory subsystem design. First, the basic motivation for a candidate design is discussed and each component of the proposed design is described. Then, an analytic model is developed to estimate the effective memory access time of the memory subsystem as a function of its design parameters. These parameters are then chosen to minimize the effective memory access time for the symbolic workload locality characteristics. Finally, the performance of the memory subsystem is evaluated by comparing it to a conventional design on the Explorer II.

7.1 Motivation for the Design

One of the key characteristics of symbolic workloads that is not exploited directly by conventional memory subsystem designs is the structural locality of the memory references. As was shown in Chapter 5, the best prediction for the next memory reference is the same simulated LRU stack position which was last referenced. If the simulated LRU stack can be realized in a cache with full associativity, then any position in the stack can be referenced in constant time, and the references immediately above the stack position accessed can be prefetched into a smaller faster on-chip cache which I have named the structural locality cache (SLC).

There are two concerns with using the SLC. First, it requires that the main cache be fully-associative. Typically, CAM requires four times the circuitry of con-

ventional memory [Blau87]. Any CAM design, then, should also perform comparably to a conventional memory system design with a direct-mapped cache four times as large. The other concern is cache pollution¹ in the CAM caused by prefetching from the main memory. The structural locality present in the LRU stack occurred when no prefetching was employed. The larger the blocks prefetched, the less mapping of temporal to spatial locality in the CAM, and hence the less structural locality.

7.2 Proposed Design

As shown in Figure 7.1, the proposed design is a three-level memory hierarchy consisting of a large physical memory (MEM), a much smaller main CAM cache (CAM), and the very small on-chip SLC. The CAM's purpose is to emulate the top portion of the simulated LRU stack thus mapping temporal locality into spatial locality within the CAM. The SLC is then able to use the structural locality captured in the CAM to prefetch memory references.

Because the CAM is finite, it cannot, as a rule, contain the entire simulated LRU stack for a given workload. If a straight LRU replacement algorithm was used, the CAM would have to be reordered after each access. The primary objective, though, of the CAM is to enable prefetching by the SLC based on structural locality. This objective can be met using a FIFO circular buffer replacement algorithm. This replacement algorithm can be expected to result in a slightly higher miss ratio, however, since even the most frequently referenced memory locations are guaranteed to be periodically removed from the CAM. On the other hand, since the SLC is much smaller than the CAM and does not have the same requirement for complete ordering, an LRU replacement algorithm is preferable for this smaller cache.

¹Smith defines cache pollution as the proportion of the cache taken up by prefetched data which is never referenced during its stay in the cache.

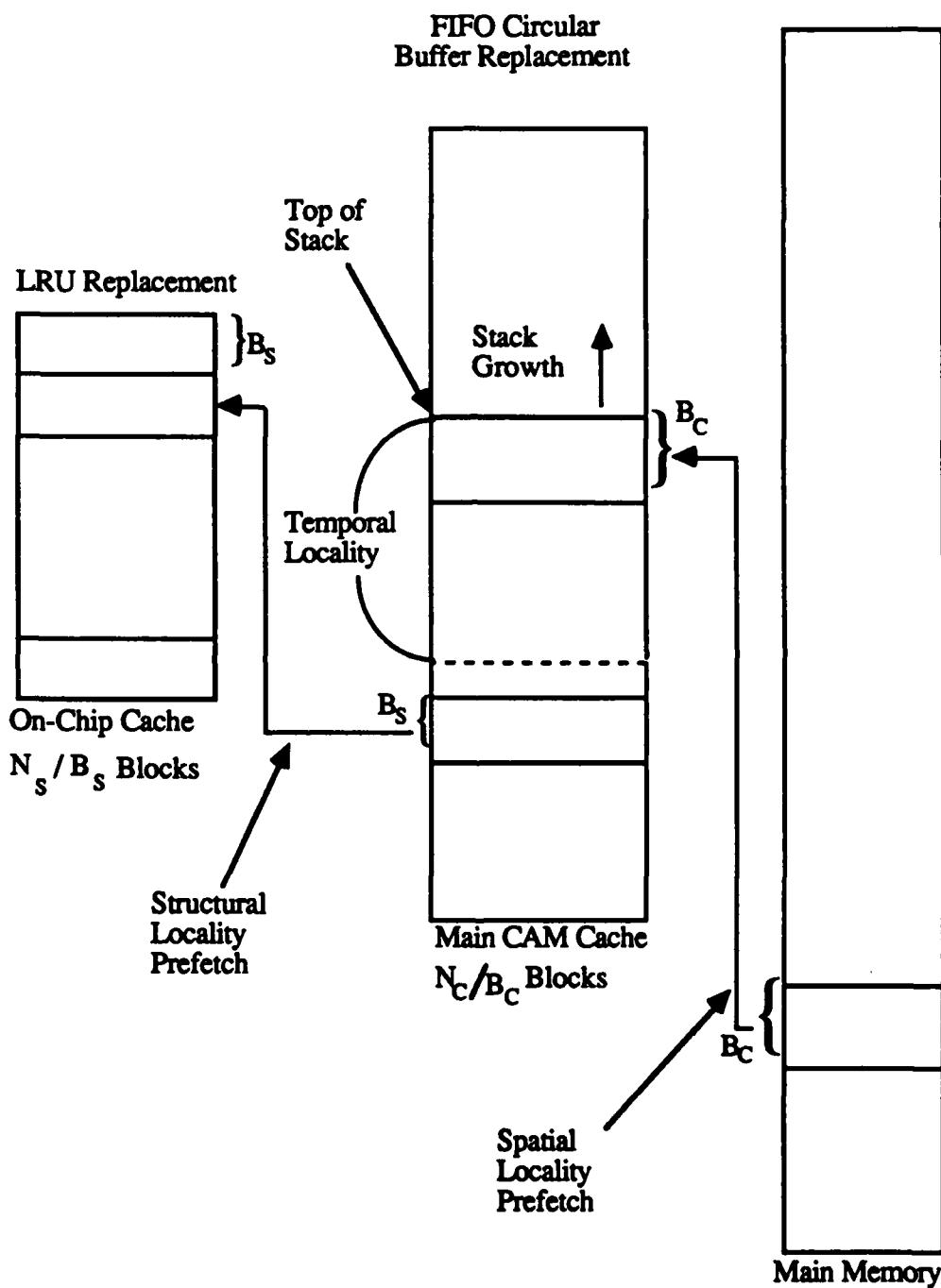


Figure 7.1: Proposed Memory Subsystem Design

Table 7.1: Memory Subsystem Design Parameters

<u>Symbol</u>	<u>Description</u>	<u>Locality Characteristics</u>
N_C	Size in words of CAM	LRU_{95}, LRU_{99}
B_C	Number of words prefetched to CAM	P_{NN}, P_{SW}
N_S	Size in words of SLC	LRU_{95}, LRU_{99}
B_S	Number of words prefetched to SLC	P_{SSD}

7.3 Specific Design Parameters

The design parameters to be chosen for this implementation and the locality characteristics on which they are primarily dependent are shown in Table 7.1. Belady has shown that the CAM miss rate cannot be guaranteed to decrease monotonically with increasing CAM size when a FIFO replacement algorithm is used [Bela69b]. However, in most cases, the CAM miss rate should decrease as its size is increased. Therefore, to arrive at maximum limits for N_C and N_S , a reasonable starting point is the current memory subsystem design of the Explorer II. The Explorer II has a cache of 32K words, and so using Blauuw's four to one size ratio, the corresponding size for a fully-associative cache would be 8K words. Assuming that some of the chip area used for the PDL could be reallocated for the SLC constrains the size of N_S , since allocating too large a percentage of the PDL chip area would increase memory traffic due to PDL overflows [Mitc86].

7.4 Analytic Model of Effective Memory Access Time

The following model estimates the effective memory access time from the workload locality characteristics and cache design parameters in Table 7.1. The

effective memory access time can first be computed as

$$t_a = t_{SLC} p_{SLC} + t_{CAM}(1 - p_{SLC})p_{CAM} + t_{MEM}(1 - p_{CAM})$$

where

t_{SLC} is the memory access time of the SLC

p_{SLC} is the hit rate of the SLC

t_{CAM} is the memory access time of the CAM

p_{CAM} is the hit rate of the CAM and

t_{MEM} is the memory access time of the main memory

This can be rewritten as

$$t_n = p_{SLC} + R_{CAM}(1 - p_{SLC})p_{CAM} + R_{MEM}(1 - p_{CAM})$$

where

$$t_n = t_a / t_{SLC}$$

$$R_{CAM} = t_{CAM} / t_{SLC} \text{ and}$$

$$R_{MEM} = t_{MEM} / t_{SLC}$$

7.4.1 CAM Cache Hit Probability and Design Parameters

The probability of a CAM cache hit, p_{CAM} , will be addressed first. If no prefetching takes place, then using the model first developed in Chapter 4 and the state transition probabilities shown in Figure 7.2, the expected CAM cache hit rate can be calculated for the workload locality characteristics measured in Chapter 5.

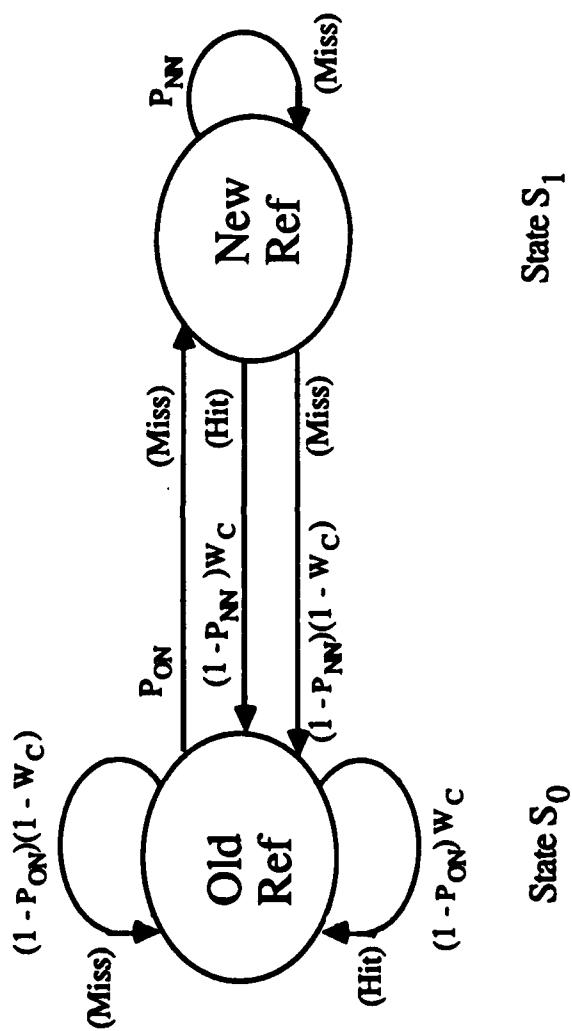


Figure 7.2: Markov Model for Main CAM Cache Referencing with No Prefetching

State S_0 is the state where the reference is to a previously-referenced memory location. And likewise, state S_1 is the state where the reference is to a new memory reference location.

The state transition probabilities can be estimated using the measured locality characteristics. The New-New transition probability, P_{NN} , can then be used to estimate the likelihood of remaining in S_1 and the New-Old transition probability, or $1 - P_{NN}$, can be used to estimate the likelihood of transitioning from S_1 to S_0 . However, because the size of the CAM cache is limited, a transition from S_1 to S_0 does not assure a cache hit. The weighting factor, W_c , uses the temporal lifetime function derived from the temporal locality cumulative histograms to estimate the likelihood of a cache hit for this transition. There were no significant differences between the temporal locality histograms of the overall, New-Old, and Old-NSSD distance strings in their overall shape nor their 90th, 95th, and 99th stack distance thresholds. Thus, the overall temporal locality cumulative histogram was used in lieu of using the individual temporal locality histograms for the New-Old and Old-NSSD distance strings. In addition, the temporal locality characteristics for the eight symbolic workloads executed on the Explorer II were averaged to generate the function of W_c versus stack distance shown in Figure 7.3. Finally, the Old-New transition probability, P_{ON} , is used to estimate the likelihood of transitioning from S_0 to S_1 and $1 - P_{ON}$ to estimate the likelihood of remaining in S_0 . Again, W_c is used to estimate the cache hit rate for the transition loop on S_0 . The cache hit rate can then be computed as

$$p_{CAM} = p_{S_0}(1 - P_{ON})W_c + p_{S_1}(1 - P_{NN})W_c$$

where

$$p_{S_0} = \frac{1 - P_{NN}}{P_{ON} + (1 - P_{NN})}$$

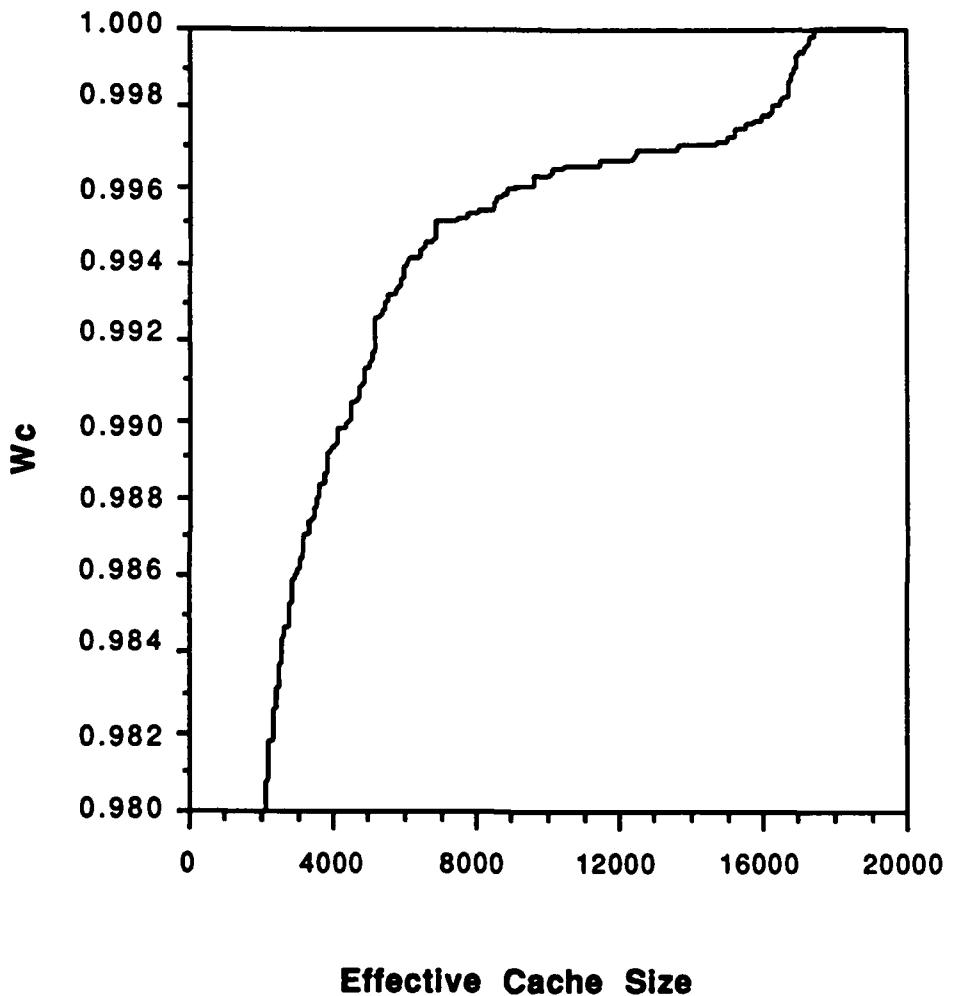


Figure 7.3: W_c vs. Effective Cache Size

and

$$p_{S_1} = 1 - p_{S_0}$$

Substituting for p_{S_0} and p_{S_1} gives

$$p_{CAM} = \frac{W_c(1 - P_{NN})}{P_{ON} + (1 - P_{NN})}$$

From Figure 7.3, the value of W_c corresponding to an 8 Kword LRU stack is 0.996. Then using the mean values of P_{ON} and P_{NO} for the symbolic workloads from Table A-11 in Appendix A², the value of p_{CAM} is computed as

$$p_{CAM} = \frac{(0.996)(0.371)}{0.013 + 0.371} = 0.962$$

As discussed in the Chapter 5, the individual frequency histograms and the values of P_{NN} for the symbolic workloads indicate that the prefetch size should be kept small. Also, since analysis of the individual frequency spatial distance histograms shows that neither a fetch forward nor a fetch backward strategy is optimal for a preponderance of the symbolic workloads, a prefetch of plus or minus one reference from the cache miss should be used. Since care must also be taken not to prefetch a reference already in the CAM cache, a good approximation to this prefetch strategy is to prefetch the block of four words in which the reference occurs, or in other words, prefetch using all but the two least significant bits of the address.

The average number of references used within this four-word prefetched block during the New-New referencing mode was found to be 1.41 using a probability decision tree and summing the expected value of each of the outcomes. Thus, on the average only 0.41 out of the additional 3 prefetched references in the CAM cache is actually referenced during the New-New referencing behavior, effectively reducing

² $P_{NO} = 1 - P_{NN}$

the cache size to 1.41/4 or 35 percent that of the CAM cache when prefetching is not used because of the unreferenced prefetches.

As shown in Figure 7.4, the New-New referencing of the prefetched block can be modeled as a loop on state S_0 with a transition probability, p_{CP} , chosen such that the expected value of the number of New-New references, U_{CP} , is equal to the 0.41 additional references calculated from the decision tree. The expected value of number of New-New prefetch references is

$$U_{CP} = (1 - p_{CP}) \sum_{i=1}^{\infty} i p_{CP}^i = (1 - p_{CP}) \frac{p_{CP}}{(1 - p_{CP})^2} = \frac{p_{CP}}{1 - p_{CP}}$$

where p_{CP} is the New-New CAM prefetch block reference probability. Now, setting the expected value of U_{CP} equal to 0.41 additional references calculated from the decision tree gives

$$\frac{p_{CP}}{1 - p_{CP}} = 0.41 \Rightarrow p_{CP} = 0.291$$

The CAM cache hit probability then is

$$p_{CAM} = (1 - P_{ON})W_c p_{S_0} + ((1 - P_{NN})W_c + p_{CP})p_{S_1}$$

Substituting for p_{S_0} and p_{S_1} gives

$$p_{CAM} = \frac{W_c(1 - P_{NN}) + p_{CP}P_{ON}}{P_{ON} + (1 - P_{NN})}$$

Calculating the new value of W_c for a cache size of $(1.41/4)8192 = 2888$ words from Figure 7.3 gives $W_c = 0.986$ and so

$$p_{CAM} = \frac{(0.986)(0.371) + (0.291)(0.013)}{0.013 + 0.371} = 0.962$$

Thus, the reduction in effective cache size due to the unused prefetches offsets the increase in cache hits from the prefetched block, and the cache hit probability is not improved by the prefetching strategy.

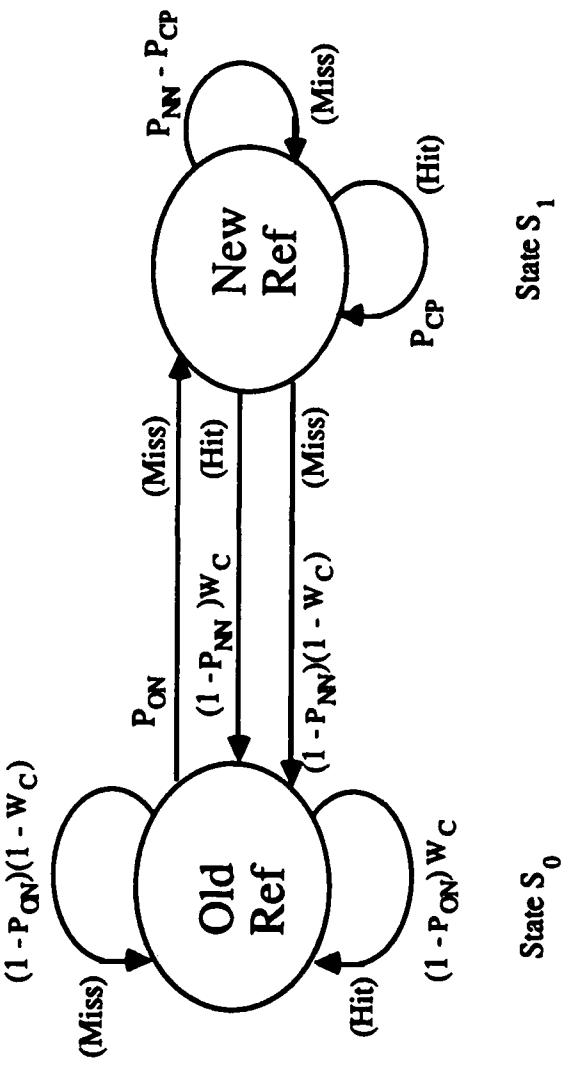


Figure 7.4: Markov Model for Main CAM Cache Referencing with Prefetching

Since there is no advantage to prefetching and since unused references would decrease the structural locality that is captured spatially in the CAM cache, the proposed CAM cache design is that no prefetching take place, or in other words, that $B_C = 1$.

7.4.2 Structural Locality Cache Hit Probability and Design Parameters

The SLC hit probability will first be calculated with no prefetching. While the primary objective of the CAM cache is to exploit structural locality through prefetching into the SLC, the hit rate with no prefetching provides a basis for evaluating the effectiveness of structural locality prefetching.

With no prefetching, the SLC model is almost identical to the CAM cache model as shown in Figure 7.5. The only exception is the use of W_s in place of W_c . The SLC hit probability is then

$$p_{SLC} = \frac{W_s(1 - P_{NN})}{P_{ON} + (1 - P_{NN})}$$

where W_s is a function of N_S as shown in Figure 7.6.

Prefetching can be incorporated into the SLC model as it was for the CAM cache. As can be seen from Figure 7.7, an SLC hit can occur when the reference is to the same stack position or to a recently referenced location. Since only B_S references will be in the prefetch block from the CAM cache, any strings of SSD references greater than B_S will cause SLC misses. To compensate for the finite prefetch block size, an adjusted SSD transition probability, P_{SSD_A} , must be used so that the expected value of the consecutive SLC hits due to SSD references, U_{SPM} , is equal to the expected value given the prefetch block size. The expected value of the model's SSD reference strings is

$$U_{SPM} = (1 - P_{SSD_A}) \sum_{i=1}^{\infty} i P_{SSD_A}^i = (1 - P_{SSD_A}) \frac{P_{SSD_A}}{(1 - P_{SSD_A})^2} = \frac{P_{SSD_A}}{1 - P_{SSD_A}}$$

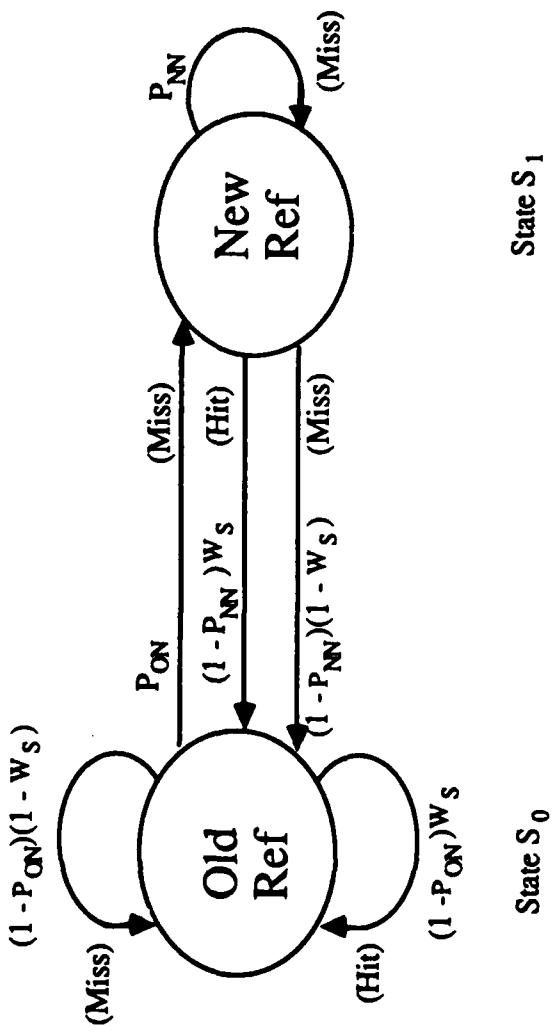


Figure 7.5: Markov Model for On-Chip SLC Referencing with No Prefetching

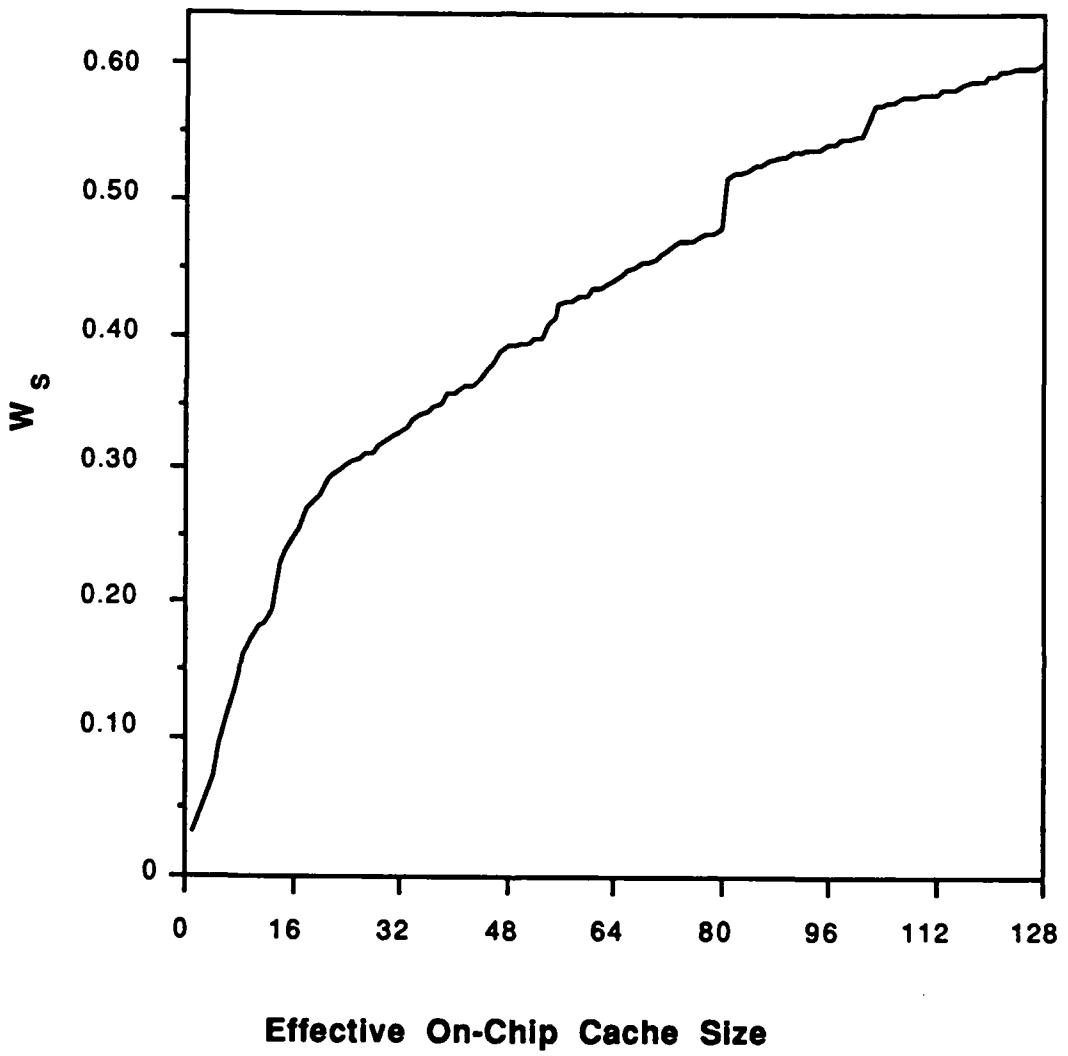


Figure 7.6: W_s vs. Effective On-Chip SLC Cache Size

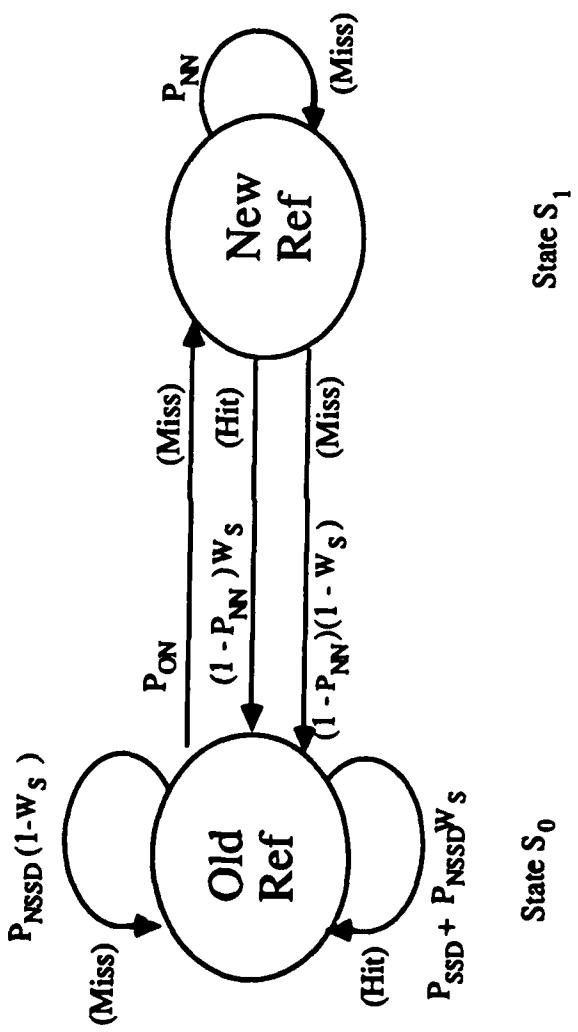


Figure 7.7: Markov Model for On-Chip SLC Referencing with Prefetching

and the expected value of the prefetch block references, U_{SP} , is

$$\begin{aligned} U_{SP} &= (1 - P_{SSD}) \sum_{i=1}^{B_S-1} i P_{SSD}^i \\ &= (1 - P_{SSD}) \frac{P_{SSD}}{(1 - P_{SSD})^2} ((B_S - 1) P_{SSD} B_S - B_S P_{SSD} B_{S-1} + 1) \\ &= \frac{P_{SSD}}{1 - P_{SSD}} ((B_S - 1) P_{SSD} B_S - B_S P_{SSD} B_{S-1} + 1) \end{aligned}$$

Setting $U_{SPM} = U_{SP}$ and solving for the adjusted P_{SSD} , P_{SSDA} , gives

$$P_{SSDA} = \frac{U_{SP}}{1 + U_{SP}}$$

There is also another adjustment to make. The effective size of the SLC cache will be reduced by references in the prefetch block which are not accessed before they are swapped out. Thus, W_s will decrease as B_S increases. To calculate the effective size of the cache, the actual cache size is reduced by the expected value of the percentage of each prefetch block which is unreferenced. The unreferenced portion of the prefetch block can be thought of as a form of cache pollution and will be denoted by p_{SP} . The value of p_{SP} is calculated as

$$p_{SP} = \frac{(B_S - 1) - (1 - P_{SSD}) \sum_{i=1}^{B_S-1} i P_{SSD}^i}{B_S} = \frac{(B_S - 1) - U_{SP}}{B_S}$$

Using Figure 7.6, W_s can be calculated for the effective cache size which is $N_S(1 - p_{SP})$. With these adjustments for the prefetching it is now possible to calculate the SLC hit probability.

$$p_{SLC} = (P_{SSDA} + P_{NSSD} W_s) p_{S_0} + (1 - P_{NN}) W_s p_{S_1}$$

Solving for p_{S_0} and p_{S_1} and substituting gives

$$p_{SLC} = \frac{1 - P_{NN}}{P_{ON} + (1 - P_{NN})} (P_{SSDA} + W_s (P_{NSSD} + P_{ON}))$$

Table 7.2: Structural Locality Cache Hit Probability (p_{SLC})

N_S	128	64	32	16	8	4	2	1
B_S								
1	0.579	0.424	0.317	0.240	0.145	0.070	0.038	0.029
2	0.393	0.342	0.303	0.261	0.230	0.212	0.202	
4	0.572	0.535	0.492	0.458	0.436	0.424		
8	0.646	0.603	0.567	0.543	0.529			
16	0.620	0.583	0.559	0.545				

Using the measured locality symbolic workload locality characteristics from Table A-11 in Appendix A and the above equation, the SLC hit probability varies with prefetch block size as shown in Table 7.2 for a range of values of N_S . The values of B_S which maximize p_{SLC} for a given N_S are highlighted in **boldface**.

7.5 Performance Analysis

Using the CAM cache and SLC hit probabilities and typical values of $R_{CAM} = 4$ and $R_{MEM} = 32$, it is now possible to estimate the effective access time for this memory subsystem design. Substituting these values into the previous equation for an SLC size of 8 words and a prefetch block size of 8 words gives

$$t_n = 0.529 + (1 - 0.529)(4)(0.962) + (32)(1 - 0.962) = 3.557 \text{ clock cycles}$$

Similarly, Table 7.3 gives other effective memory access times for $R_{CAM} = 4$ and $R_{MEM} = 32$ and for values of N_S ranging from 2 to 128 words and using the values of B_S highlighted in Table 7.2. To compare this with a conventional memory design without a CAM cache, p_{CAM} must be computed for a cache four times as large. From Figure 7.3, W_c for a 32 Kword cache is 0.997. Also, for this cache size,

Table 7.3: Performance Analysis for $R_{CAM} = 4$ and $R_{MEM} = 32$

N_S	128	64	32	16	8	4	2
t_n	3.224	3.347	3.449	3.518	3.557	3.856	4.489
t_b	3.082	3.530	3.839	4.062	4.337	4.554	4.646
$S (\%)$	-4.4	5.5	11.3	15.5	21.9	18.1	3.5

prefetching does result in a lower cache miss rate. So, computing the new p_{CAM} for the conventional cache with $B_C = 4$ gives

$$p_{CAM} = \frac{W_c(1 - P_{NN}) + p_{CP}P_{ON}}{P_{ON} + (1 - P_{NN})} = \frac{(0.997)(0.371) + (0.291)(0.013)}{0.013 + 0.371} = 0.973$$

Therefore, the base effective memory access time for the conventional memory design using prefetching is

$$t_b = 0.145 + (1 - 0.145)(4)(0.973) + (32)(1 - 0.973) = 4.337 \text{ clock cycles}$$

The speedup (given in percent) is then defined as

$$S = \left(\frac{t_b}{t_n} - 1 \right) 100$$

Thus, for an on-chip structural locality cache of 8 words, a system CAM cache memory access time of 4 clock cycles, and a main memory access time of 32 clock cycles, this design is about 22 percent faster than the conventional hierarchical memory design used as a comparison as shown below:

$$S = \left(\frac{4.337}{3.557} - 1 \right) 100 = 21.9 \text{ percent}$$

Table 7.3 gives other effective memory access times, base comparison times, and the effective speedup for $R_{CAM} = 4$ and $R_{MEM} = 32$ and for values of N_S ranging from 2 to 128 words using the values of B_S highlighted in Table 7.2.

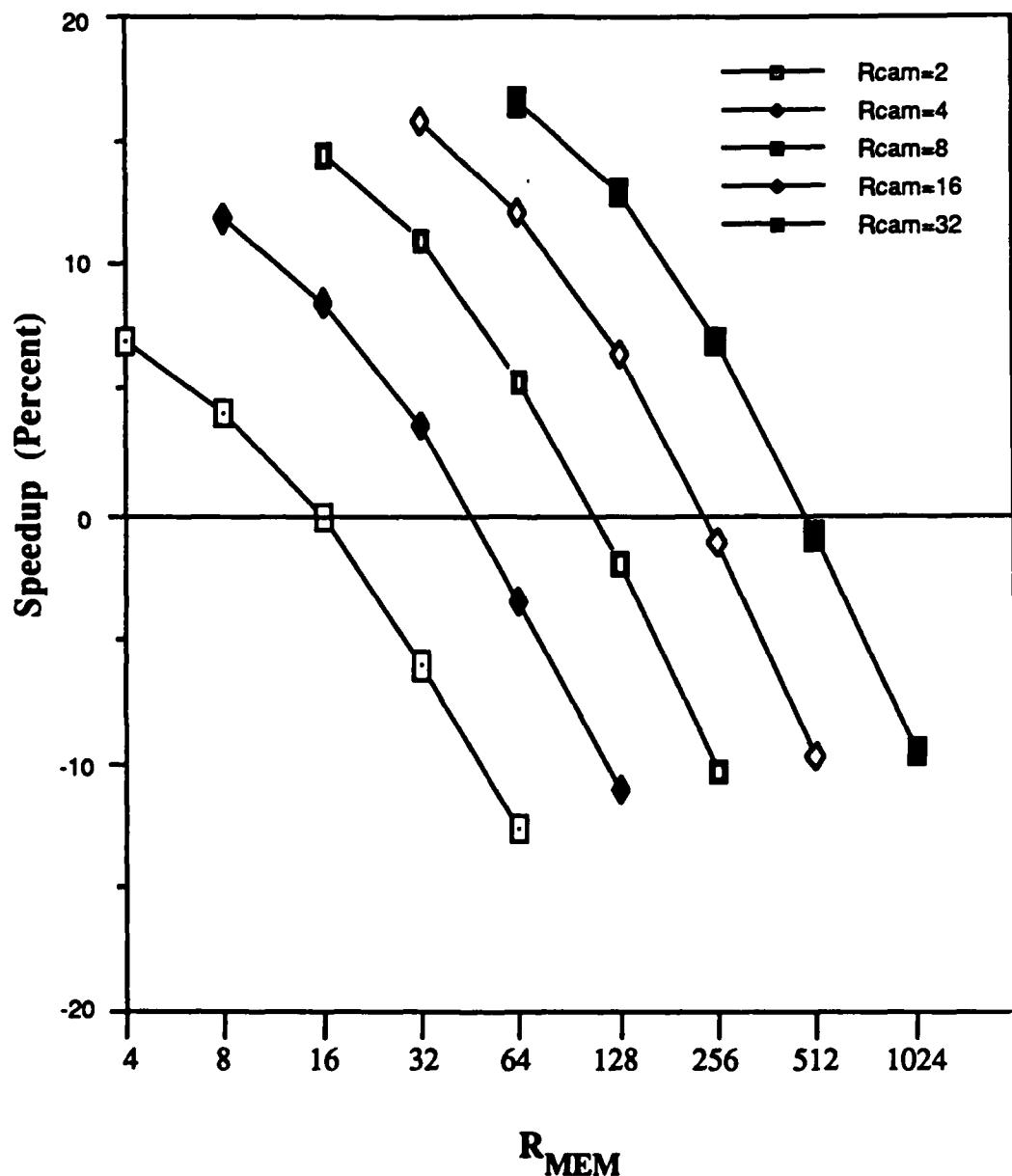


Figure 7.8: Speedup vs. R_{CAM} and R_{MEM} for $N_S = 2$ words

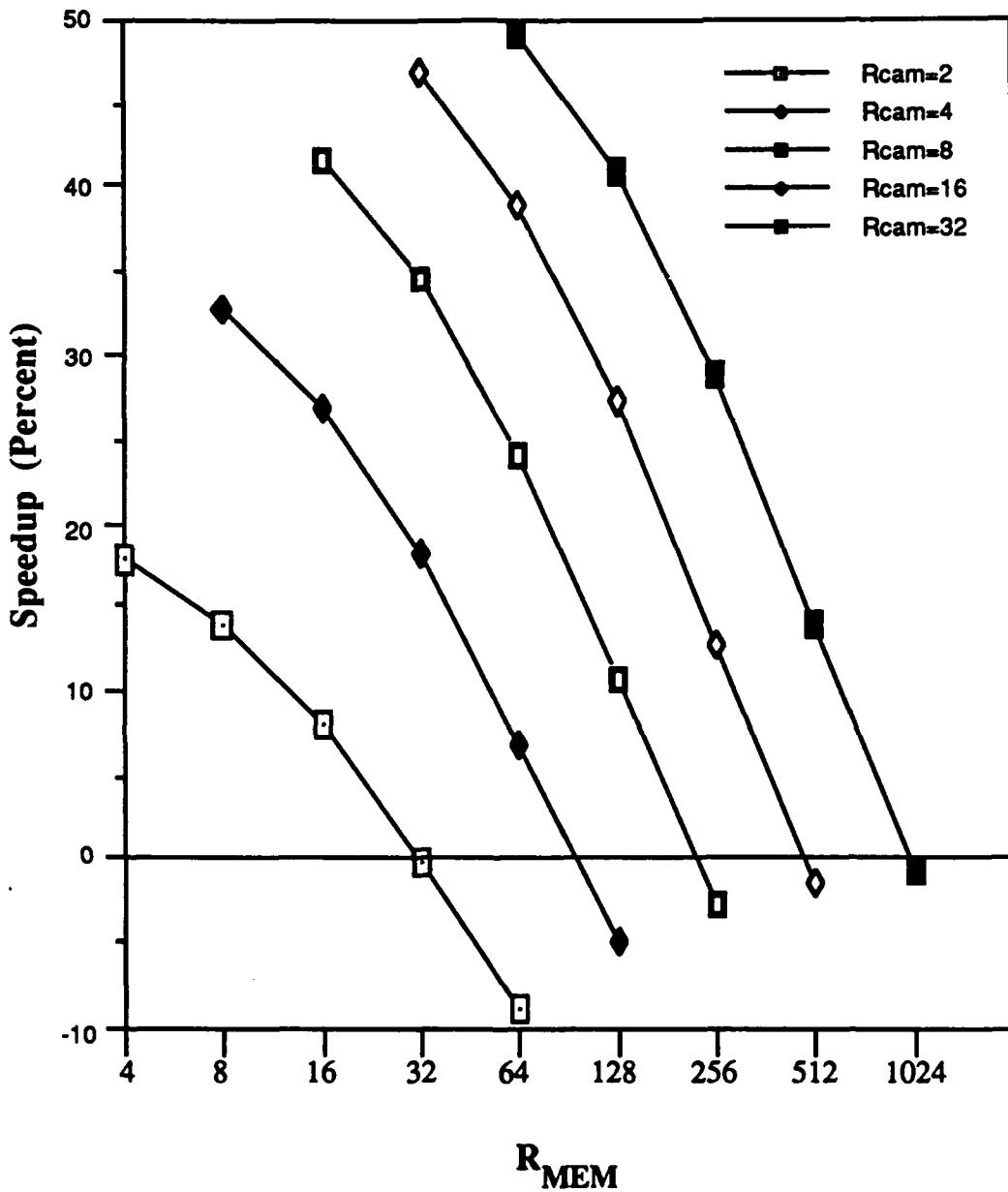


Figure 7.9: Speedup vs. R_{CAM} and R_{MEM} for $N_S = 4$ words

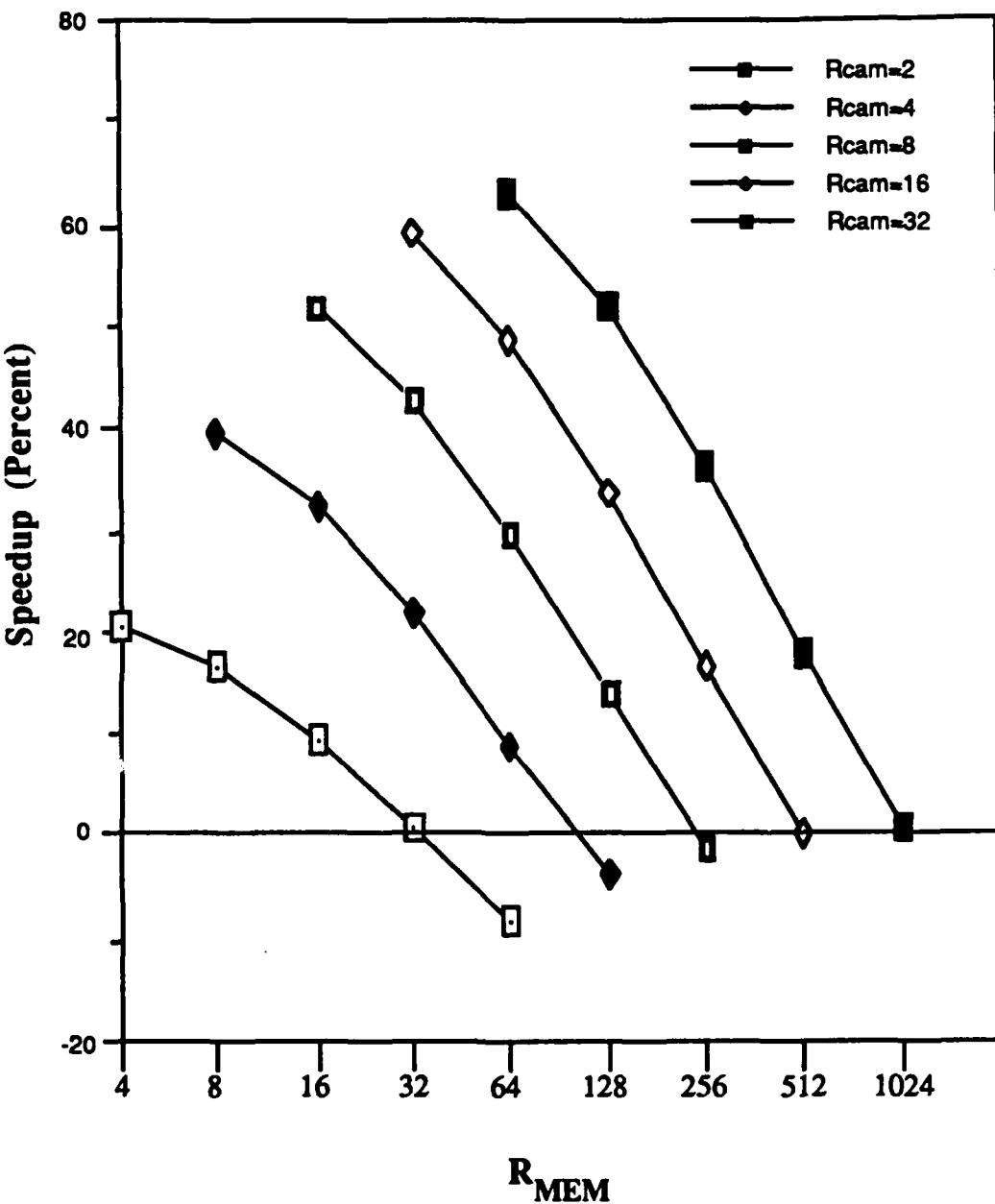


Figure 7.10: Speedup vs. R_{CAM} and R_{MEM} for $N_S = 8$ words

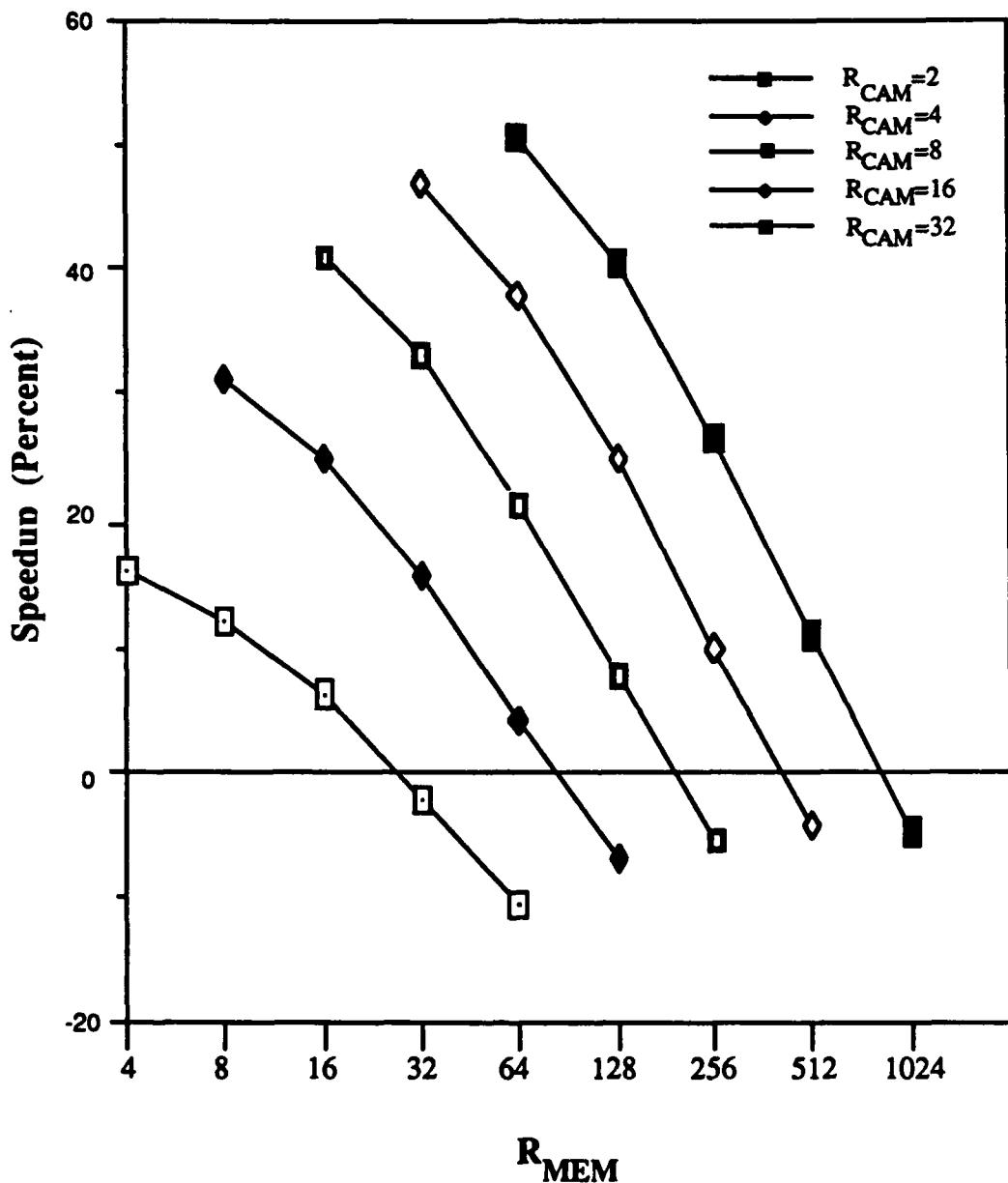


Figure 7.11: Speedup vs. R_{CAM} and R_{MEM} for $N_S = 16$ words

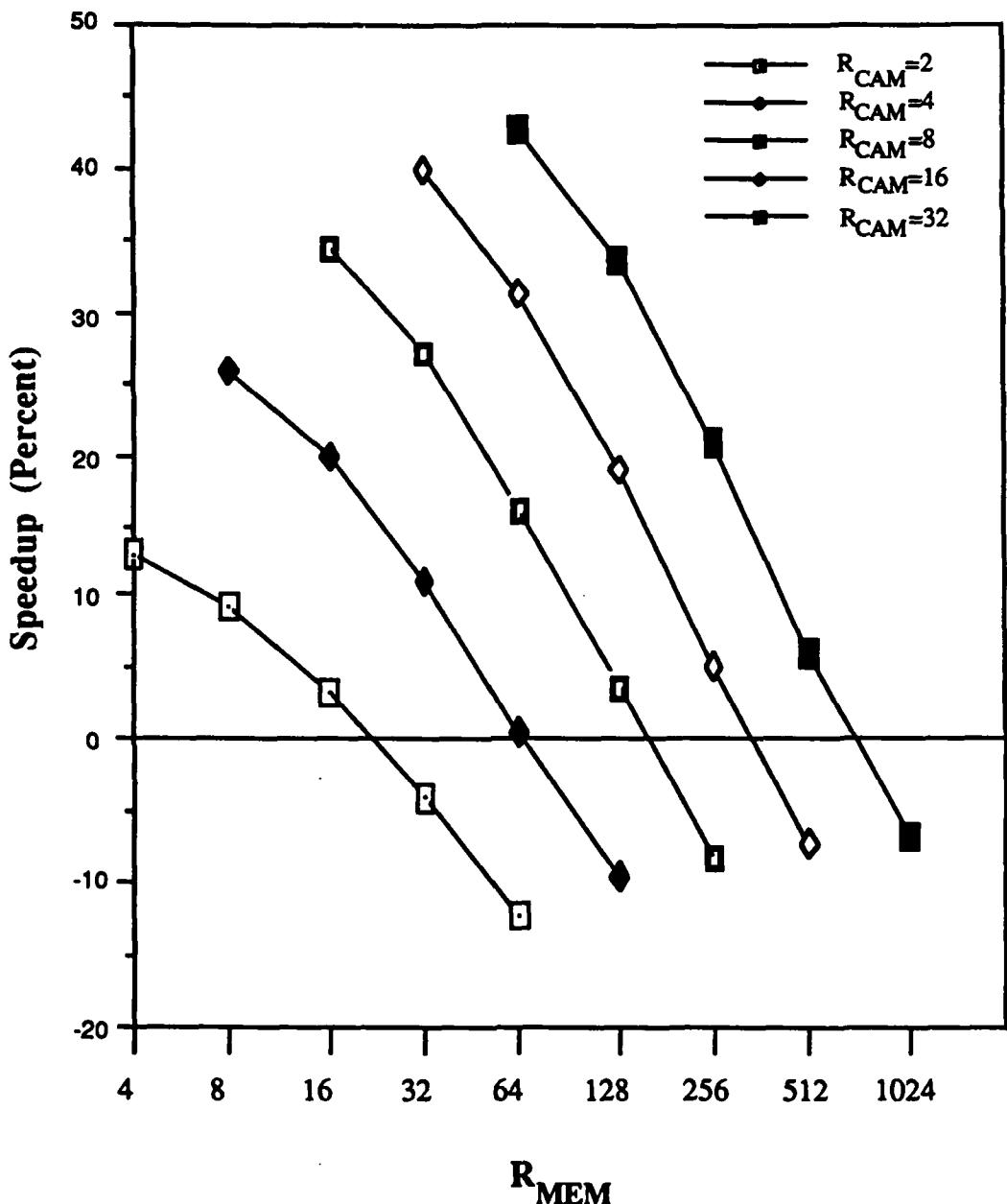


Figure 7.12: Speedup vs. R_{CAM} and R_{MEM} for $N_S = 32$ words

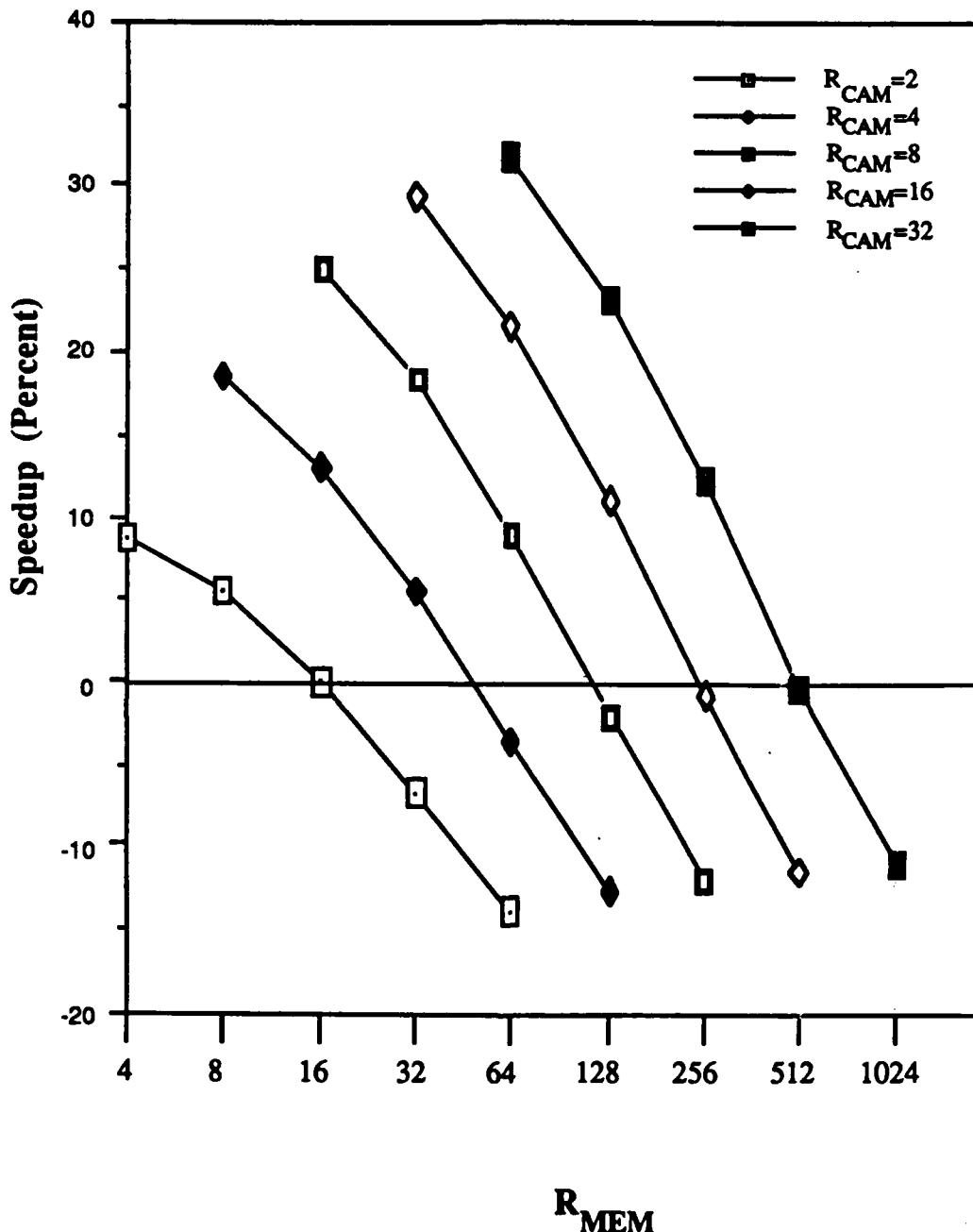


Figure 7.13: Speedup vs. R_{CAM} and R_{MEM} for $N_S = 64$ words

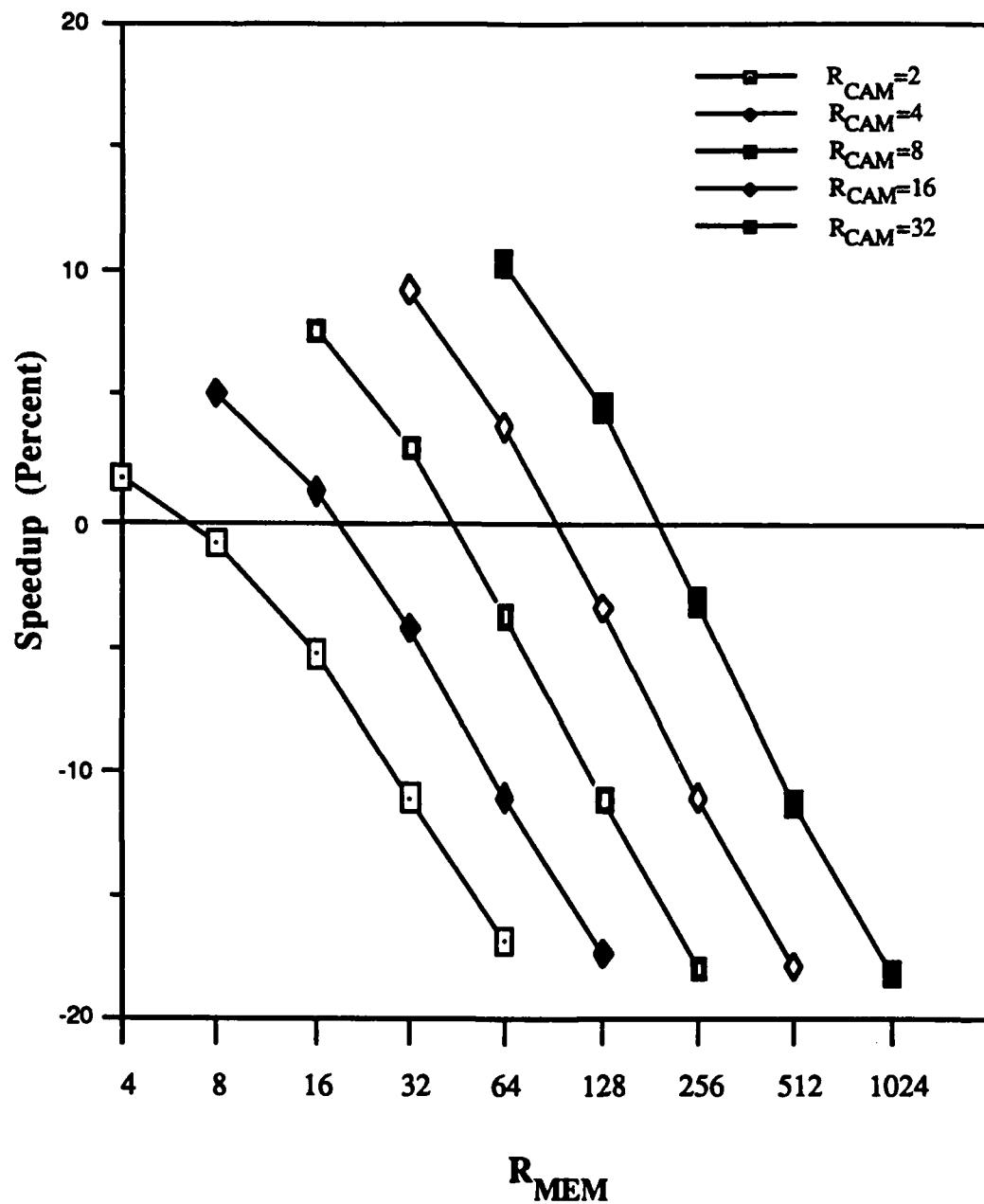


Figure 7.14: Speedup vs. R_{CAM} and R_{MEM} for $N_S = 128$ words

Figures 7.8 through 7.14 show the speedups attainable with this architecture for a range of values of R_{CAM} and R_{MEM} . For all values of R_{CAM} and R_{MEM} , the highest speedup, although not necessarily the minimum access time, occurs when N_S and B_S are equal to 8 words. For larger on-chip cache sizes, the temporal locality tends to capture a large portion of the Old-SSD transitions, thus decreasing the benefits of a design tailored for structural locality prefetching.

7.6 Summary

This chapter has shown how the results of this research can be used to design a memory subsystem tailored to the special structural locality characteristics of symbolic workloads. It also shows how possible designs such as the one in this chapter using a structural locality cache can be quantitatively evaluated and how the design's proper regime for application can be identified.

Chapter 8

Conclusion

8.1 Main Contributions

The principal contribution of this research is the identification of significant differences in the virtual word-level memory addressing behavior of symbolic and conventional workloads. These differences in the spatial, temporal, and structural locality of symbolic and conventional workloads provide direction to memory designers tailoring their designs for a particular application. As shown in Chapter 7, the locality characteristics of symbolic workloads can be used as an input to a systematic design process producing a memory design with the minimum effective access time for that type of workload.

The metrics and experimental methodology developed for this research are also significant contributions. Especially significant is the identification of a low-level measure of structural locality and the relating of this locality to the Same-Stack-Distance phenomenon observed in the virtual memory address traces. While structural locality has been previously acknowledged, no other published measurements so directly characterize this type of locality.

Nor do other published characterizations suggest the direct exploitation of structural locality evaluated in Chapter 7. This novel design option is another contribution of this research. While it does not guarantee lower access times for all technologies, as the penalty for accessing memory off-chip increases and the larger main cache sizes cause their access times to more closely approach those of main

memory, this design option will become more attractive.

Finally, the extension of Denning's page reference Markov model to virtual word-level memory addressing behavior contributes to the development of symbolic workload generators which, in turn, can be used to provide input to expert system design tools. For example, an adaptation of the Markov model presented in this dissertation is already in use as part of a cache memory design tool [Rim89].

8.2 Additional Applications of This Research

There are several ways in which this research can be extended. First, as the integrated circuit complexity increases, there is greater opportunity for building into the cache controller a monitoring system, similar to those proposed by Belady in [Bela73] and [Bela74] for paging control, which allows cache parameters such as the prefetch block size or caching strategy to periodically adapt to measured locality characteristics. The results detailed in Chapter 5 suggest another possible approach. A memory system design could be dynamically tailored for conventional or symbolic workloads by first classifying the memory referencing behavior using the locality characteristics, and then adopting the appropriate cache control strategy. Thus this research provides a sound footing for the investigation of dynamic cache control.

Another application is to incorporate the results of this research into models of the multitasking operating system environment. This would involve an extension to the Markov model to represent task switching behavior and an architecturally independent characterization of operating system behavior. Techniques similar to those used by Wong and Morris in their benchmark synthesis model based on their LRU hit function have potential for extending symbolic workload characterization for these environments [Wong88].

Still another application would be to use the locality metrics and experi-

mental methodology developed in this research to characterize the low-level virtual memory referencing behavior of various garbage collection algorithms. This would provide valuable insight into the design of memory systems tailored for symbolic workloads where garbage collection is a major concern.

There are also other design alternatives which can be evaluated using the models developed in this research and similar methodology to that in Chapter 7. For example, the ability of a split cache for instruction and data references to exploit the distinct locality characteristics of the instruction fetches could be evaluated and its performance compared to that of a unified cache.

Possibly, the greatest potential for building upon this research lies in developing a unifying theory of program behavior which explains and interrelates the spatial, temporal, and structural aspects of locality of reference characterized by this research. When one considers the small standard deviations of the values of many of the locality metrics computed in this research for a wide range of programming styles, data structure usage, and applications there appears to be an underlying pattern of virtual address memory referencing behavior that transcends all these factors. One possible approach, would be to pursue further the 'order' content of the memory reference string with methods similar to those of Hammerstrom and Davidson [Hamm77]. Specifically, analyzing the information theoretic content or entropy of the spatial and temporal distance strings sorted by the type of reference (e.g., instruction fetch) and the transition type (e.g., New-New) could yield new insight. Using this approach, it may thus be possible to achieve an understanding of program behavior which has proven so elusive over the past quarter century.

In summary, there are many ways to apply the results of this research to memory system design. What this research has done is to lay a foundation based on a systematic experimental process and measurable locality characteristics for the

analysis of low-level virtual memory referencing behavior. And hopefully, by quantifying differences between symbolic and conventional workload memory referencing locality, this research has helped to make the memory subsystem design process more systematic.

Appendix A

Individual Workload Locality Measurements

Table A-1. Spatial Locality -- All References**CPU Type: Explorer II****Workload Type: Symbolic**

Workload	$P_{sw\text{-All}}$	$P_{sw\text{-Old-New}}$	$P_{sw\text{-New-New}}$
Boyer	0.452	0.011	0.954
Compile-RB	0.467	0.262	0.689
Compile-STR	0.463	0.264	0.708
GLISP-Comp	0.527	0.141	0.741
GLISP-Pay	0.443	0.188	0.877
QSIM	0.460	0.131	0.825
Reducer	0.525	0.135	0.965
TMYCIN	0.616	0.076	0.901
Mean for Wkld Type:	0.494	0.151	0.832
Std Dev for Wkld Type:	0.0587	0.0866	0.1092
Workload Type: Conventional			
BIASLisp	0.314	0.028	0.940
FFT-Gabriel	0.350	0.004	0.839
Mean for Wkld Type:	0.332	0.016	0.889
Std Dev for Wkld Type:	0.0255	0.017	0.0714
Mean for CPU Type:	0.461	0.124	0.843
Std Dev for CPU Type:	0.0862	0.0954	0.1021
CPU Type: IBM System/360 Model 91			
Workload Type: Conventional			
APL-Plotter	0.519	0.611	0.943
FFT1	0.345	0.110	0.915
FFT2	0.351	0.110	0.911
WATEX-BinPack	0.390	0.337	0.460
WATFIV-Comp	0.397	0.495	0.623
Mean for Wkd/CPU Type:	0.400	0.332	0.770
Std Dev for Wkd/CPU Type:	0.0702	0.2253	0.2171
Overall Mean:	0.441	0.193	0.819
Overall Std Dev:	0.0841	0.1752	0.1465

Table A-2. Spatial Locality -- Instruction Fetches**CPU Type: Explorer II****Workload Type: Symbolic**

Workload	$P_{sw\text{-All}}$	$P_{sw\text{-Old-New}}$	$P_{sw\text{-New-New}}$
Boyer	0.962	0.917	0.927
Compile-RB	0.862	0.721	0.930
Compile-STR	0.878	0.721	0.930
GLISP-Comp	0.893	0.723	0.905
GLISP-Pay	0.902	0.689	0.933
QSIM	0.906	0.694	0.895
Reducer	0.976	0.670	0.930
TMYCIN	0.880	0.741	0.909
Mean for Wkld Type:	0.907	0.734	0.919
Std Dev for Wkld Type:	0.0407	0.0772	0.0146
Workload Type: Conventional			
BIASLisp	0.915	0.812	0.944
FFT-Gabriel	0.999	0.833	0.933
Mean for Wkld Type:	0.957	0.822	0.938
Std Dev for Wkld Type:	0.0594	0.0149	0.0078
Mean for CPU Type:	0.917	0.752	0.923
Std Dev for CPU Type:	0.0460	0.0777	0.0153

CPU Type: IBM System/360 Model 91**Workload Type: Conventional**

APL-Plotter	0.932	0.929	0.827
FFT1	0.885	0.899	0.815
FFT2	0.886	0.894	0.819
WATEX-BinPack	0.907	0.685	0.820
WATFIV-Comp	0.914	0.875	0.812
Mean for Wkld/CPU Type:	0.904	0.856	0.818
Std Dev for Wkld/CPU Type:	0.0198	0.0978	0.0057
Overall Mean:	0.913	0.786	0.888
Overall Std Dev:	0.0389	0.0959	0.0528

Table A-3. Spatial Locality -- All Data References**CPU Type: Explorer II****Workload Type: Symbolic**

Workload	P_{SW} -All	P_{SW} -Old-New	P_{SW} -New-New
Boyer	0.215	0.011	0.954
Compile-RB	0.376	0.291	0.697
Compile-STR	0.393	0.292	0.727
GLISP-Comp	0.430	0.126	0.718
GLISP-Pay	0.333	0.177	0.894
QSIM	0.560	0.148	0.819
Reducer	0.561	0.085	0.966
TMYCIN	0.637	0.062	0.902
Mean for Wkld Type:	0.438	0.149	0.834
Std Dev for Wkld Type:	0.1395	0.1019	0.1095
Workload Type: Conventional			
BIASLisp	0.321	0.027	0.962
FFT-Gabriel	0.336	0.004	0.808
Mean for Wkld Type:	0.328	0.015	0.885
Std Dev for Wkld Type:	0.0106	0.0163	0.1089
Mean for CPU Type:	0.416	0.122	0.844
Std Dev for CPU Type:	0.1315	0.1062	0.1053

CPU Type: IBM System/360 Model 91

Workload Type: Conventional	P_{SW} -All	P_{SW} -Old-New	P_{SW} -New-New
APL-Plotter	0.562	0.412	0.984
FFT1	0.309	0.032	0.971
FFT2	0.309	0.031	0.972
WATEX-BinPack	0.373	0.142	0.608
WATFIV-Comp	0.538	0.232	0.792
Mean for Wkld/CPU Type:	0.418	0.169	0.865
Std Dev for Wkld/CPU Type:	0.1234	0.1593	0.1645
Overall Mean:	0.416	0.138	0.851
Overall Std Dev:	0.1244	0.1226	0.1223

Table A-4. Spatial Locality -- Data Reads**CPU Type: Explorer II****Workload Type: Symbolic**

Workload	P_{SW-All}	$P_{SW-Old-New}$	$P_{SW-New-New}$
Boyer	0.178	0.467	0.696
Compile-RB	0.331	0.314	0.642
Compile-STR	0.351	0.335	0.684
GLISP-Comp	0.406	0.261	0.670
GLISP-Pay	0.285	0.175	0.643
QSIM	0.562	0.385	0.744
Reducer	0.492	0.055	0.720
TMYCIN	0.645	0.163	0.788
Mean for Wkld Type:	0.406	0.269	0.698
Std Dev for Wkld Type:	0.1531	0.1336	0.0505
Workload Type: Conventional			
BIASLisp	0.287	0.388	0.862
FFT-Gabriel	0.304	0.381	0.680
Mean for Wkld Type:	0.295	0.384	0.771
Std Dev for Wkld Type:	0.0120	0.005	0.1287
Mean for CPU Type:	0.384	0.292	0.712
Std Dev for CPU Type:	0.1429	0.1275	0.069

CPU Type: IBM System/360 Model 91**Workload Type: Conventional**

APL-Plotter	0.530	0.251	0.984
FFT1	0.385	0.108	0.962
FFT2	0.385	0.107	0.960
WATEX-BinPack	0.356	0.095	0.550
WATFIV-Comp	0.543	0.216	0.714
Mean for Wkld/CPU Type:	0.439	0.155	0.834
Std Dev for Wkld/CPU Type:	0.0892	0.0725	0.1935
Overall Mean:	0.402	0.246	0.753
Overall Std Dev:	0.1271	0.1281	0.1314

Table A-5. Spatial Locality -- Data Writes**CPU Type: Explorer II****Workload Type: Symbolic**

Workload	$P_{SW\text{-All}}$	$P_{SW\text{-Old-New}}$	$P_{SW\text{-New-New}}$
Boyer	0.342	0.000	0.998
Compile-RB	0.443	0.057	0.958
Compile-STR	0.460	0.048	0.956
GLISP-Comp	0.377	0.022	0.969
GLISP-Pay	0.627	0.007	0.982
QSIM	0.427	0.001	0.739
Reducer	0.896	0.004	0.989
TMYCIN	0.461	0.011	0.986
Mean for Wkld Type:	0.504	0.018	0.947
Std Dev for Wkld Type:	0.1791	0.0221	0.0854
Workload Type: Conventional			
BIASLisp	0.230	0.003	0.916
FFT-Gabriel	0.174	0.001	0.799
Mean for Wkld Type:	0.202	0.002	0.857
Std Dev for Wkld Type:	0.0396	0.0014	0.0827
Mean for CPU Type:	0.443	0.015	0.929
Std Dev for CPU Type:	0.2034	0.0207	0.0887

CPU Type: IBM System/360 Model 91

Workload Type: Conventional	$P_{SW\text{-All}}$	$P_{SW\text{-Old-New}}$	$P_{SW\text{-New-New}}$
APL-Plotter	0.759	0.464	0.996
FFT1	0.573	0.027	0.147
FFT2	0.573	0.023	0.148
WATEX-BinPack	0.450	0.168	0.621
WATFIV-Comp	0.577	0.287	0.844
Mean for Wkld/CPU Type:	0.586	0.193	0.551
Std Dev for Wkld/CPU Type:	0.1105	0.1867	0.3919
Overall Mean:	0.491	0.074	0.803
Overall Std Dev:	0.1869	0.1335	0.2880

Table A-6. Temporal Locality -- All References**CPU Type: Explorer II****Workload Type: Symbolic**

Workload	LRU ₉₀	LRU ₉₅	LRU ₉₉
Boyer	221	838	3822
Compile-RB	467	845	3234
Compile-STR	1020	2197	16777
GLISP-Comp	840	1405	4390
GLISP-Pay	1638	1673	1694
QSIM	532	1460	4158
Reducer	517	806	2570
TMYCIN	634	1038	2400
Mean for Wkld Type:	733	1282	4880
Std Dev for Wkld Type:	437	494	4895
Workload Type: Conventional			
BIASLisp	1473	25276	25987
FFT-Gabriel	2645	29137	29199
Mean for Wkld Type:	2059	27206	27593
Std Dev for Wkld Type:	828	2730	2271
Mean for CPU Type:	998	6467	9423
Std Dev for CPU Type:	733	10976	10531

CPU Type: IBM System/360 Model 91**Workload Type: Conventional**

APL-Plotter	71	250	3380
FFT1	118	119	184
FFT2	118	119	184
WATEX-BinPack	136	306	532
WATFIV-Comp	802	995	1882
Mean for Wkld/CPU Type:	249	357	1232
Std Dev for Wkld/CPU Type:	310	365	1389
Overall Mean:	748	4430	6692
Overall Std Dev:	712	9294	9371

Table A-7. Temporal Locality -- Instruction Fetches**CPU Type: Explorer II****Workload Type: Symbolic**

Workload	LRU₉₀	LRU₉₅	LRU₉₉
Boyer	30	33	50
Compile-RB	270	454	1314
Compile-STR	563	1223	6742
GLISP-Comp	465	656	1196
GLISP-Pay	753	753	756
QSIM	134	592	635
Reducer	72	156	650
TMYCIN	291	509	631
Mean for Wkld Type:	322	547	1496
Std Dev for Wkld Type:	253	366	2154
Workload Type: Conventional			
BIASLisp	363	363	557
FFT-Gabriel	29	29	40
Mean for Wkld Type:	196	196	298
Std Dev for Wkld Type:	236	236	365
Mean for CPU Type:	297	476	1257
Std Dev for CPU Type:	243	364	1969

CPU Type: IBM System/360 Model 91**Workload Type: Conventional**

APL-Plotter	36	52	452
FFT1	82	82	110
FFT2	82	82	110
WATEX-BinPack	61	200	358
WATFIV-Comp	530	640	1334
Mean for Wkld/CPU Type:	158	211	472
Std Dev for Wkld/CPU Type:	208	246	504
Overall Mean:	250	388	995
Overall Std Dev:	234	345	1647

Table A-8. Temporal Locality -- All Data References**CPU Type: Explorer II****Workload Type: Symbolic**

Workload	LRU₉₀	LRU₉₅	LRU₉₉
Boyer	697	1062	5132
Compile-RB	229	408	2057
Compile-STR	441	1096	9863
GLISP-Comp	463	800	3480
GLISP-Pay	910	922	943
QSIM	561	869	3384
Reducer	474	732	1812
TMYCIN	427	575	1726
Mean for Wkld Type:	519	788	3538
Std Dev for Wkld Type:	205	247	2880
Workload Type: Conventional			
BIASLisp	13909	24786	25527
FFT-Gabriel	16485	29040	29092
Mean for Wkld Type:	15197	26913	27309
Std Dev for Wkld Type:	1821	3008	2520
Mean for CPU Type:	3454	6013	8293
Std Dev for CPU Type:	6220	11062	10373

CPU Type: IBM System/360 Model 91**Workload Type: Conventional**

APL-Plotter	205	388	4685
FFT1	37	58	81
FFT2	37	58	80
WATEX-BinPack	111	143	314
WATFTV-Comp	290	366	696
Mean for Wkld/CPU Type:	136	202	1171
Std Dev for Wkld/CPU Type:	110	163	1980
Overall Mean:	2348	4076	5919
Overall Std Dev:	5244	9312	9076

Table A-9. Temporal Locality -- Data Reads

CPU Type: Explorer II

Workload Type: Symbolic

Workload	LRU ₉₀	LRU ₉₅	LRU ₉₉
Boyer	539	735	2662
Compile-RB	236	426	2216
Compile-STR	474	1145	9649
GLISP-Comp	463	852	3428
GLISP-Pay	853	862	881
QSIM	605	867	3359
Reducer	499	829	2204
TMYCIN	427	600	1572
Mean for Wkld Type:	512	789	3246
Std Dev for Wkld Type:	174	211	2723
Workload Type: Conventional			
BIASLisp	23716	24167	25001
FFT-Gabriel	24390	27002	27037
Mean for Wkld Type:	24053	25584	26019
Std Dev for Wkld Type:	476	2004	1439
Mean for CPU Type:	5220	5748	7800
Std Dev for CPU Type:	9928	10477	9909

CPU Type: IBM System/360 Model 91

Workload Type: Conventional

APL-Plotter	227	354	4108
FFT1	34	58	66
FFT2	34	58	66
WATEX-BinPack	121	165	321
WATFIV-Comp	297	353	652
Mean for Wkld/CPU Type:	142	197	1042
Std Dev for Wkld/CPU Type:	117	148	1730
Overall Mean:	3527	3898	5548
Overall Std Dev:	8337	8826	8651

Table A-10. Temporal Locality -- Data Writes

CPU Type: Explorer II

Workload Type: Symbolic

Workload	LRU ₉₀	LRU ₉₅	LRU ₉₉
Boyer	8	13	31
Compile-RB	44	142	1442
Compile-STR	142	458	2816
GLISP-Comp	142	365	2899
GLISP-Pay	110	143	143
QSIM	82	95	750
Reducer	81	85	1456
TMYCIN	100	171	803
Mean for Wkld Type:	88	184	1292
Std Dev for Wkld Type:	46	150	1095
Workload Type: Conventional			
BLASLisp	21700	21952	22096
FFT-Gabriel	24911	24919	24935
Mean for Wkld Type:	23305	23435	23515
Std Dev for Wkld Type:	2270	2097	2007
Mean for CPU Type:	4732	4834	5737
Std Dev for CPU Type:	9818	9829	9443

CPU Type: IBM System/360 Model 91

Workload Type: Conventional

APL-Plotter	1315	4185	4388
FFT1	42	50	50
FFT2	42	50	50
WATEX-BinPack	71	117	151
WATFIV-Comp	133	162	439
Mean for Wkld/CPU Type:	320	912	1015
Std Dev for Wkld/CPU Type:	557	1829	1891
Overall Mean:	3261	3527	4163
Overall Std Dev:	8166	8168	7978

Table A-11. State Transition Probabilities -- All References**CPU Type: Explorer II****Workload Type: Symbolic**

Workload	P _{NO}	P _{SSD}	P _{NSSD}	P _{ON}
Boyer	0.506	0.474	0.502	0.024
Compile-RB	0.382	0.562	0.423	0.015
Compile-STR	0.383	0.544	0.438	0.018
GLISP-Comp	0.478	0.623	0.361	0.016
GLISP-Pay	0.250	0.588	0.407	0.005
QSIM	0.457	0.444	0.544	0.012
Reducer	0.137	0.540	0.454	0.006
TMYCIN	0.378	0.626	0.364	0.010
Mean for Wkld Type:	0.371	0.550	0.436	0.013
Std Dev for Wkld Type:	0.1235	0.0653	0.0634	0.0063
Workload Type: Conventional				
BIASLisp	0.706	0.269	0.677	0.054
FFT-Gabriel	0.685	0.316	0.619	0.065
Mean for Wkld Type:	0.695	0.292	0.648	0.059
Std Dev for Wkld Type:	0.0149	0.0332	0.0410	0.0078
Mean for CPU Type:	0.436	0.498	0.478	0.022
Std Dev for CPU Type:	0.1748	0.1234	0.1061	0.0205

CPU Type: IBM System/360 Model 91**Workload Type: Conventional**

APL-Plotter	0.315	0.126	0.869	0.005
FFT1	0.560	0.161	0.837	0.003
FFT2	0.558	0.160	0.837	0.003
WATEX-BinPack	0.751	0.101	0.896	0.004
WATFIV-Comp	0.749	0.161	0.832	0.007
Mean for Wkld/CPU Type:	0.586	0.141	0.854	0.004
Std Dev for Wkld/CPU Type:	0.1794	0.0273	0.0276	0.0017
Overall Mean:	0.486	0.379	0.604	0.016
Overall Std Dev:	0.1850	0.2008	0.2025	0.0187

Table A-12. State Transition Probabilities -- Instruction Fetches**CPU Type: Explorer II****Workload Type: Symbolic**

Workload	P _{NO}	P _{SSD}	P _{NSSD}	P _{ON}
Boyer	0.236	0.830	0.170	0.000
Compile-RB	0.113	0.954	0.044	0.002
Compile-STR	0.117	0.944	0.054	0.002
GLISP-Comp	0.152	0.929	0.070	0.001
GLISP-Pay	0.099	0.919	0.081	0.000
QSIM	0.158	0.896	0.103	0.001
Reducer	0.196	0.975	0.024	0.001
TMYCIN	0.113	0.933	0.067	0.000
Mean for Wkld Type:	0.148	0.922	0.076	0.000
Std Dev for Wkld Type:	0.0478	0.0441	0.0446	0.0008
Workload Type: Conventional				
BIASLisp	0.075	0.985	0.015	0.000
FFT-Gabriel	0.062	0.999	0.001	0.000
Mean for Wkld Type:	0.068	0.992	0.008	0.000
Std Dev for Wkld Type:	0.0092	0.0099	0.0099	0.000
Mean for CPU Type:	0.132	0.936	0.062	0.000
Std Dev for CPU Type:	0.0539	0.0488	0.0489	0.0008

CPU Type: IBM System/360 Model 91**Workload Type: Conventional**

APL-Plotter	0.855	0.477	0.518	0.004
FFT1	0.783	0.477	0.522	0.000
FFT2	0.780	0.475	0.524	0.000
WATEX-BinPack	0.876	0.506	0.492	0.003
WATFIV-Comp	0.825	0.534	0.461	0.005
Mean for Wkld/CPU Type:	0.823	0.493	0.503	0.002
Std Dev for Wkld/CPU Type:	0.0427	0.0259	0.027	0.0023
Overall Mean:	0.362	0.788	0.209	0.001
Overall Std Dev:	0.3410	0.2199	0.219	0.0016

Table A-13. State Transition Probabilities -- Data References**CPU Type: Explorer II****Workload Type: Symbolic**

Workload	P _{NO}	P _{S SSD}	P _{N SSD}	P _{ON}
Boyer	0.506	0.627	0.349	0.024
Compile-RB	0.454	0.653	0.336	0.011
Compile-STR	0.445	0.654	0.333	0.013
GLISP-Comp	0.534	0.746	0.241	0.013
GLISP-Pay	0.259	0.714	0.282	0.004
QSIM	0.477	0.808	0.181	0.010
Reducer	0.121	0.722	0.274	0.005
TMYCIN	0.393	0.744	0.247	0.009
Mean for Wkld Type:	0.398	0.708	0.280	0.011
Std Dev for Wkld Type:	0.1403	0.0603	0.0575	0.0062
Workload Type: Conventional				
BIASLisp	0.714	0.359	0.588	0.053
FFT-Gabriel	0.674	0.389	0.547	0.064
Mean for Wkld Type:	0.694	0.374	0.567	0.058
Std Dev for Wkld Type:	0.0283	0.0212	0.029	0.0078
Mean for CPU Type:	0.457	0.641	0.337	0.020
Std Dev for CPU Type:	0.1758	0.1509	0.1316	0.0209

CPU Type: IBM System/360 Model 91**Workload Type: Conventional**

APL-Plotter	0.109	0.820	0.178	0.001
FFT1	0.547	0.756	0.242	0.002
FFT2	0.546	0.757	0.241	0.002
WATEX-BinPack	0.684	0.744	0.255	0.002
WATFIV-Comp	0.673	0.733	0.264	0.003
Mean for Wkld/CPU Type:	0.511	0.762	0.236	0.002
Std Dev for Wkld/CPU Type:	0.2347	0.0339	0.0338	0.0007
Overall Mean:	0.475	0.681	0.303	0.014
Overall Std Dev:	0.1905	0.1357	0.1180	0.0190

Table A-14. State Transition Probabilities -- Data Reads**CPU Type: Explorer II****Workload Type: Symbolic**

Workload	P _{NO}	P _{SSD}	P _{NSSD}	P _{ON}
Boyer	0.892	0.708	0.260	0.032
Compile-RB	0.480	0.781	0.208	0.011
Compile-STR	0.455	0.764	0.223	0.013
GLISP-Comp	0.555	0.821	0.167	0.013
GLISP-Pay	0.571	0.760	0.235	0.005
QSIM	0.569	0.841	0.147	0.011
Reducer	0.893	0.803	0.162	0.035
TMYCIN	0.630	0.826	0.163	0.012
Mean for Wkld Type:	0.630	0.788	0.195	0.016
Std Dev for Wkld Type:	0.1706	0.0437	0.0414	0.0108
Workload Type: Conventional				
BIASLisp	0.687	0.678	0.273	0.050
FFT-Gabriel	0.653	0.721	0.221	0.058
Mean for Wkld Type:	0.670	0.699	0.247	0.054
Std Dev for Wkld Type:	0.0240	0.0304	0.0368	0.0057
Mean for CPU Type:	0.638	0.770	0.205	0.024
Std Dev for CPU Type:	0.1516	0.0546	0.0442	0.0186

CPU Type: IBM System/360 Model 91**Workload Type: Conventional**

APL-Plotter	0.071	0.859	0.140	0.001
FFT1	0.128	0.840	0.160	0.000
FFT2	0.129	0.840	0.159	0.000
WATEX-BinPack	0.666	0.813	0.186	0.001
WATFIV-Comp	0.662	0.855	0.143	0.002
Mean for Wkld/CPU Type:	0.331	0.841	0.157	0.000
Std Dev for Wkld/CPU Type:	0.3047	0.0181	0.0183	0.0008
Overall Mean:	0.536	0.794	0.189	0.016
Overall Std Dev:	0.2526	0.0567	0.0436	0.0187

Table A-15. State Transition Probabilities -- Data Writes**CPU Type: Explorer II****Workload Type: Symbolic**

Workload	P_{NO}	P_{SSD}	P_{NSSD}	P_{ON}
Boyer	0.500	0.966	0.012	0.022
Compile-RB	0.369	0.966	0.031	0.003
Compile-STR	0.330	0.964	0.032	0.004
GLISP-Comp	0.574	0.965	0.027	0.008
GLISP-Pay	0.133	0.980	0.019	0.002
QSIM	0.480	0.985	0.008	0.006
Reducer	0.080	0.994	0.003	0.003
TMYCIN	0.402	0.958	0.035	0.006
Mean for Wkld Type:	0.358	0.972	0.020	0.006
Std Dev for Wkld Type:	0.1742	0.0125	0.0121	0.0065
Workload Type: Conventional				
BIASLisp	0.669	0.838	0.117	0.045
FFT-Gabriel	0.698	0.824	0.119	0.057
Mean for Wkld Type:	0.683	0.831	0.118	0.051
Std Dev for Wkld Type:	0.0205	0.0099	0.0014	0.0085
Mean for CPU Type:	0.423	0.944	0.040	0.015
Std Dev for CPU Type:	0.206	0.0607	0.0423	0.0197

CPU Type: IBM System/360 Model 91**Workload Type: Conventional**

APL-Plotter	0.013	0.972	0.028	0.000
FFT1	0.472	0.962	0.037	0.001
FFT2	0.471	0.962	0.036	0.001
WATEX-BinPack	0.563	0.939	0.060	0.001
WATFIV-Comp	0.724	0.882	0.000	0.002
Mean for Wkld/CPU Type:	0.448	0.943	0.032	0.001
Std Dev for Wkld/CPU Type:	0.2645	0.0364	0.0216	0.0007
Overall Mean:	0.431	0.943	0.037	0.010
Overall Std Dev:	0.2177	0.0524	0.0361	0.0173

BIBLIOGRAPHY

- [Alex75] Alexander, W. G. and D. B. Wortman, "Static and Dynamic Characteristics of XPL Programs," *Computer* 8 (11) pp. 41-46 (November 1975).
- [Agar86] Agarwal, A., R. L. Sites, and M. Horowitz, "ATUM: A New Technique for Capturing Address Traces Using Microcode," *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 119-127 (June 1986).
- [Bela66] Belady, L. A., "A Study of Replacement Algorithms for a Virtual Storage Computer," *IBM Systems Journal* 5 (2) pp. 78-101 (February 1966).
- [Bela69a] Belady, L. A. and C. J. Kuehner, "Dynamic Space-Sharing in Computer Systems," *Communications of the ACM* 12 (5) pp. 282-288 (May 1969).
- [Bela69b] Belady, L. A., R. A. Nelson, and G. S. Shedler, "An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine," *Communications of the ACM* 12 (6) pp. 349-353 (June 1969).
- [Beia73] Belady, L. A. and R. F. Tsao, "Memory Allocation and Program Behavior Under Multiprogramming," *Proceedings of the 7th Annual Interface Symposium on Computer Science and Statistics*, pp. 72-78 (October 1973).
- [Bela74] Belady, L. A. and F. P. Palermo, "On-line Measurement of Paging Behavior by the Multivalued MIN Algorithm," *IBM Journal of Research and Development* 18 (1) pp. 2-19 (January 1974). [Blau87] Blaauw, G. A.

- and F. P. Brooks, Jr., Computer Architecture, Vol 1—Design Decisions, (Draft, Unpublished 1987).
- [Clar77] Clark, D. W. and C. C. Green, "An Empirical Study of List Structure in Lisp," *Communications of the ACM* 20 (2) pp. 78- 87 (February 1977).
- [Clar79] Clark, D. W., "Measurements of Dynamic List Structure in Lisp," *IEEE Transactions on Software Engineering* 5 (1) pp. 51-59 (January 1979).
- [Clar85] Clark, D. W. and J. S. Emer, "Performance of the VAX- 11/780 Translation Buffer: Simulation and Measurement," *ACM Transactions on Computer Systems* 3 (1) pp. 31-62 (February 1985).
- [Denn68] Denning, P. J., "The Working Set Model for Program Behavior," *Communications of the ACM* 11 (5) pp. 323-333 (May 1968).
- [Denn72] Denning, P. J., "On Modeling Program Behavior," *Proceedings of the Spring Joint Computer Conference*, pp. 937-944 (1972).
- [Emer84] Emer, J. S. and D. W. Clark, "A Characterization of Processor Performance in the VAX-11/780," *Proceedings of the 11th International Symposium on Computer Architecture*, pp. 301-310 (June 1984).
- [Flyn87] Flynn, M. J., C. L. Mitchell, and J. M. Mulder, "And Now a Case for More Complex Instruction Sets," *Computer* 20 (9) pp. 71- 83 (September 1987).
- [Fode81] Foderaro, J. K. and R. J. Fateman, "Characterization of VAX Macsyma," *Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation*, pp. 14-19 (August 1981).

- [Good83] Goodman, J. R., "Using Cache Memory to Reduce Processor-Memory Traffic," *Proceedings of the 10th International Symposium on Computer Architecture*, pp. 124-131 (June 1983).
- [Good84] Goodman, J. R. and M. Chiang, "The Use of Static Column RAM as a Memory Hierarchy," *Proceedings of the 11th International Symposium on Computer Architecture*, pp. 167-174 (June 1984).
- [Good86] Goodman, J. R. and W. Hsu, "On the Use of Registers vs. Cache to Minimize Memory Traffic," *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 375-383 (June 1986).
- [Gree84] Greenblatt, R. D., T. F. Knight, Jr., J. Holloway, D. A. Moon, and D. L. Weinreb, "The LISP Machine," *Interactive Programming Environments*, eds. D. R. Barstow, H. E. Shrobe, and E. Sandewall, McGraw-Hill, Hightstown, NJ, pp. 327-352 (1984).
- [Hamm77] Hammerstrom, D. W. and E. S. Davidson, "Information Content of CPU Memory Referencing Behavior," *Proceedings of the 4th International Symposium on Computer Architecture*, pp. 184-192 (March 1977)
- [Haya83] Hayashi, H. A., A. Hattori, and H. Akimoto, "Alpha: A High-Performance Lisp Machine Equipped with a New Stack Structure and Garbage Collection System," *Proceedings of the 10th International Symposium on Computer Architecture*, pp. 342-348 (June 1983).
- [Hill84] Hill, M. D. and A. J. Smith, "Experimental Evaluation of On-Chip Microprocessor Cache Memories," *Proceedings of the 11th International Symposium on Computer Architecture*, pp. 158-166 (June 1984).
- [Lee84] Lee, J. F. K. and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer* 17(1) pp. 6-22 (January 1984).

- [Lewi73] Lewis, P. A. W. and G. S. Shedler, "Empirically Derived Models for Sequences of Page Exceptions," *IBM Journal of Research Development* 17 (2) pp. 86-100 (March 1973).
- [Matt70] Mattson, R. L., J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal* 9 (2) pp. 78-117 (February 1970).
- [McNi88] McNiven, G. D. and E. S. Davidson, "Analysis of Memory Referencing Behavior for Design of Local Memories," *Proceedings of the 15th International Symposium on Computer Architecture*, pp. 56- 63 (May 1988).
- [Mitc86] Mitchell, C. L., "Processor Architecture and Cache Performance," Technical Report CSL-TR-86-296, Computer Systems Laboratory, Stanford University, Stanford, CA (June 1986).
- [Olss83] Olsson, O., "The Memory Usage of a Lisp System: the Belady Lifetime Function," *SIGPLAN Notices* 18 (12) pp.112-119 (December 1983).
- [Rees85] Reese, B. R. II, "Hardware Support for High-Level List Functions," Ph.D. Dissertation, Texas A & M University, College Station, TX (1985).
- [Rim89] Rim, Y., "A System-Level Modeling Methodology and Its Application to Multi-Cache Systems," Ph.D. Dissertation, University of Texas at Austin, Austin, TX (In Preparation).
- [Smit77] Smith, A. J., "Two Methods for the Efficient Analysis of Memory Address Trace Data," *IEEE Transactions on Software Engineering SE-3* (1) pp. 94-101 (January 1977).
- [Smit82] Smith, A. J., "Cache Memories," *ACM Computing Surveys* 14 (3) pp. 473-530 (September 1982).

- [Smit83] Smith, J. E. and J. R. Goodman, "A Study of Instruction Cache Organizations and Replacement Policies," *Proceedings of the 10th International Symposium on Computer Architecture*, pp. 132-137 (June 1983).
- [Smit85a] Smith, A. J., "Cache Evaluation and the Impact of Workload Choice," *Proceedings of the 12th International Symposium on Computer Architecture*, pp. 64-73 (June 1985).
- [Smit85b] Smith, A. J., "Disk Cache—Miss Ratio Analysis and Design Considerations," *ACM Transactions on Computer Systems* 3 (3) pp. 161-203 (August 1985).
- [Sohi85a] Sohi, G. S., E. S. Davidson, and J. H. Patel, "An Efficient Lisp-Execution Architecture with a New Representation for List Structures," *Proceedings of the 12th International Symposium on Computer Architecture*, pp. 91-98 (June 1985).
- [Sohi85b] Sohi, G. S., "BLAST: A Machine Architecture for High- Speed List Processing Using Associative Tables," Ph.D. Dissertation, University of Illinois, Urbana, IL (1985).
- [Spir77] Spirn, J. R., *Program Behavior: Models and Measurements*, Elsevier, New York, NY (1977).
- [Stan87] Stanley, T. J. and R. G. Wedig, "A Performance Analysis of Automatically Managed Top of Stack Buffers," *Proceedings of the 14th International Symposium on Computer Architecture*, pp. 272-281 (June 1987).
- [Tayl86] Taylor, G. S. and P. N. Hilfinger, J. R. Larus, D. A. Patterson, and B. G. Zorn, "Evaluation of the SPUR Lisp Architecture," *Proceedings of the 13th*

International Symposium on Computer Architecture, pp. 444-452 (June 1986)

- [Thaz86] Thazhuthaveetil, M. J., "A Structured Memory Access Architecture for Lisp," Ph.D. Dissertation, University of Wisconsin- Madison, Madison, WI (1986).
- [Wong88] Wong, W. S. and R. J. T. Morris, "Benchmark Synthesis Using the LRU Cache Hit Function," *IEEE Transactions on Computers* 37(6) pp. 637-645 (June 1988).
- [Yuhara86] Yuhara, M. et. al., "Evaluation of the FACOM Alpha Lisp Machine," *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 184-190 (June 1986).
- [Zipf49] Zipf, G. K., *Human Behavior and the Principle of Least Effort*, Addison-Wesley, Reading, MA (1949).

VITA

William Chester Hobart, Jr.

[REDACTED] After graduating from Cabrillo Senior High School, Lompoc, California, in 1972, he entered the United States Air Force Academy, Colorado Springs, Colorado. He received the degree of Bachelor of Science from the United States Air Force Academy in June, 1976, and was commissioned a Second Lieutenant in the United States Air Force. In June, 1979, after serving as a communications maintenance officer, he entered the Air Force Institute of Technology at Wright-Patterson Air Force Base, Ohio. In March, 1981, he received the degree of Master of Science in Electrical Engineering from the Air Force Institute of Technology. He then taught in the Department of Electrical Engineering at the United States Air Force Academy until June, 1983. Finally, until entering the Graduate School of the University of Texas in September, 1986, he served as Chief, Data Link Standards Branch, Tactical Air Forces Interoperability Group, Langley Air Force Base, Virginia. Upon completion of his studies, Major Hobart will join the faculty of the Air Force Institute of Technology.

[REDACTED]

This dissertation was typeset¹ with L^AT_EX by the author.

¹L^AT_EX document preparation system was developed by Leslie Lamport as a special version of Donald Knuth's T_EX program for computer typesetting. T_EX is a trademark of the American Mathematical Society. The L^AT_EX macro package for The University of Texas at Austin dissertation format was written by Khe-Sing Cheung.